AN INVESTIGATION OF REORGANIZATION ALGORITHMS

by

CHRISTOPHER ZHONG

B.S., Kansas State University, 2002

A THESIS

submitted in partial fulfillment of the
requirements for the degree

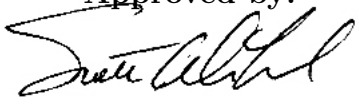
MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2006

Approved by:



Major Professor
Scott A. DeLoach

Abstract

With computer systems becoming more sophisticated, there is a need to have a high-level mechanism that is able to control these systems as a whole. Some of these systems are multiagent systems that are highly adaptable in a constantly changing environment. This thesis focuses one such approach; an organization-based system to designing and controlling agents. OMACS is designed to allow multiagent systems to be robust and flexible in a changing environment by modeling the loss and degradation of agents' capabilities. The aim of OMACS is to provide a framework to build such systems. This thesis introduces a reorganization algorithm that produces an optimal solution as the basis for future reorganization algorithms and provides some recommendations for improvements to OMACS to allow better reorganization algorithms.

In this thesis, the complexity of the reorganization algorithm that produces an optimal solution is analyzed. In a multiagent-based system, it is generally more efficient if most, if not all, of the agents contribute towards the workload of a complex problem. For this reason, a simple distributed version is also provided for analysis. In addition to analyzing the time complexity of the distributed version, a brief overview of the message complexity is provided. Furthermore, the reorganization algorithm is implemented in a high-level simulator to provide results on the practicality of the reorganization algorithm. From the test results, a number of interesting areas were uncovered. First, this thesis provides some pointers for future improvements to OMACS to allow for more efficient reorganization algorithms. Next, this thesis provides some basic metrics that can be used to evaluate their models in terms of efficiency and flexibility. And lastly, this thesis highlights some promising preliminary analysis of distributed reorganization that should be explored further.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I want to thank Dr. Scott A. DeLoach, my major professor, for his continuing support in my research. Thank you for providing me with guidance on numerous occasions during the course of my research, teaching me the mechanics of conducting research, and most importantly, thank you for pushing me to excel in my research.

I want to thank Dr. Gurdip Singh and Dr. David A. Gustafson for serving on my committee.

I want to thank my friends, Scott J. Harmon, Edwin Rodríguez, and Matt Miller for their invaluable inputs and ideas. Thank you, Scott, for taking the time to point out a number of issues with the algorithm, your help with parts of the complexity analysis, and your help in gathering some of the results. Thank you, Edwin, for providing me ideas and inputs on my research. Thank you, Matt, for your inputs on various parts of the algorithm, and OMACS, particularly the GMoDS.

Last but not least, I want to thank members of MACR research team who have helped me in one way or another during the course of my research.

Chapter 1

Introduction

As technology presses forward, more complex tasks are being delegated to computational systems. This is especially true in the area of computer science and robotics. There are even demands for dangerous tasks to be performed by robotic systems. Automation of dangerous tasks removes the risks of danger from humans. As a result, there are complex robotics systems that can perform a wide variety of tasks; tasks that humans are unable to or tasks that are too risky for humans. Generally, these systems are distributed and expected to adapt to changes in their environment. While distributed systems offer increased reliability and access to distributed resources, adaptive systems continue to perform effectively while reacting to their dynamically changing environments. However, control of such a system usually requires a team of humans. For instance, the Mars Rover robot requires a team of humans to control. With advances in computing technology, attempts are being made to reverse the trend, where instead of requiring a team of humans to control one robot, it only takes one human to control a team of robots.

One approach to building adaptive, distributed systems is that of multiagent systems. Some of the early multiagent systems are MaSE [DeL99], GAIA [WJK00], and TROPOS [BPG+04]. However, early multiagent systems were typically designed with a set of predefined goals and emphasized individual agents and their interactions. This resulted in adaptivity at the agent-level with system-level adaptation being a byproduct of the agent-level adaptation.

To achieve system-level adaptation, a system-level mechanism is required. This mechanism should define how the system-level adaptation translates into the agent-level adaptation. Such a mechanism is the focus of a number of ongoing research efforts based on an organizational metaphor. Various research groups are looking to provide mechanisms that guide a group of agents by specifying high-level objectives within a predefined organization structure. One such research is OMACS [DM05], upon which this thesis is based. Other related research groups are mentioned in § 2.1. However, only OMACS is relevant within the scope of this thesis.

1.1 Organization Model for Adaptive Computational Systems

In this section, an overview of the organization model by Dr. Scott A. DeLoach and his students is presented. Information about the model is based on three sources ([DeL05], [DM04], and [DM05]) as well as information from current ongoing research[1]. Figure 1.1 shows the organization model used in this thesis. Only a subset of the model is described because there are portions of the model that is not relevant within the scope of this thesis.

In this thesis, the organization model is referred to as the *Organization Model for Adaptive Computational Systems* (*OMACS*)[2]. OMACS defines the standard entities of an organization: goals ($\boldsymbol{G}$), roles ($\boldsymbol{R}$), and agents ($\boldsymbol{A}$). In addition, OMACS also defines three additional entities: capabilities ($\boldsymbol{C}$), assignment set ($\boldsymbol{\Phi}$), and policies ($\boldsymbol{P}$). These entities are encapsulated by an overall entity called the "Organization". Agents possesses a set of capabilities, which determine how well agents are capable of playing roles. Roles achieve a set of goals. Assignments are a tuple of an agent assigned to play a role to achieve a goal. Policies provides additional restrictions on the relationships among the entities. The definitions of these entities as well as their

[1] At the time of writing this thesis, the organization model is in the middle of transitioning to a new version. As such, most of the information in this thesis relates to the old version with some additions from the new version.

[2] This name is currently the favored name of Dr. Scott A. DeLoach's research group.

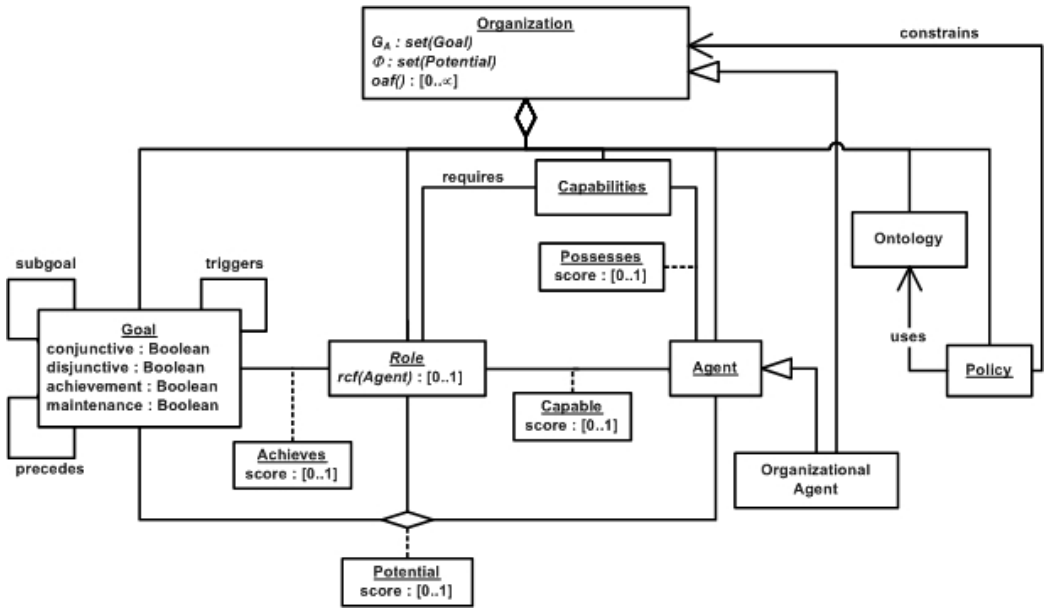


Figure 1.1: Organization Model

relationships to each other within OMACS are described in the following subsections.

In current research on OMACS, the relationships among goals, roles, and capabilities are assumed to be static. For instance, when a goal is defined as achieved by a role, that relationship cannot be changed during runtime. However, the relationship between agents and capabilities is dynamic because the dynamic relationship allows OMACS to model the degradation or loss of capabilities. Furthermore, the set of goals, roles, and capabilities are static. But the set of agents, assignments, and policies are dynamic.

1.1.1 Organization

The "Organization" entity contains a function called the `oaf()`, also known as the organization assignment function. The `oaf()` determines the effectiveness of the current assignment set. The `oaf()` returns an organizational score of a real value ranging from $0 \ldots \infty$, where the higher the organization score the better the organization performs. Typically, the `oaf()` is application specific and is usually redefined on a per application basis. However, in [DM05], a default `oaf()` is given as the sum of the

assignments from the assignment set.

$$oaf = \sum_{\forall \langle a,r,g \rangle \in \Phi} potential(a, r, g) \tag{1.1}$$

1.1.2 Goals

All organizations, even artificial organizations, have an overall goal which the organization is attempting to achieve. In OMACS[3], the overall goal is called the top-level organization goal (g_o). According to OMACS, the top-level goal can be decomposed into subgoals. Furthermore, subgoals can also be further decomposed. Decomposed subgoals describe in finer detail how the parent goal can be achieved. The goal model in OMACS is similar to the KAOS [vLLD98] approach for goal decomposition. KAOS goal based requirement modeling approach describes how goals can be decomposed into non-cyclic subgoals using either AND-refinement or OR-refinement. An AND goal can only be achieved when all its subgoals are achieved. An OR goal can be achieved when one of its subgoals is achieved.

Ensuring a proper goal structure requires three conditions: there is exactly one top-level goal, all goals (except the top-level goal) has exactly one parent, and there are no cycles in the structure. To ensure a proper goal structure, OMACS uses a single goal tree structure. The top-level goal is the root of the tree, which ensures that there is exactly one top-level goal. Furthermore, all goals (except the top-level goal) have exactly one parent goal and there are no cycles in the goal tree. The goal tree represents the set of goals the organization is trying to achieve. More specifically, since a decomposed goal can be achieved by achieving its subgoals (whether all or one depends on if the goal is an AND goal or an OR goal), only the lowest level goals needs to be achieved. In other words, the lowest level goals are goals without subgoals. OMACS defines these lowest level goals as leaf goals ($\boldsymbol{G_L}$).

Due to the generic nature of goals, goals are used in a wide variety of situations.

[3]In the new version, the goals have been moved to a goal model, *Goal Model for Dynamic Systems* (*GMoDS*), that works in conjunction with OMACS.

Thus, OMACS allows goals to be parameterized so that it is possible to determine the achievement state of a goal. For example, a goal of the payroll department is to "create paychecks". The goal "create paychecks" is so generic that the goal can be misinterpreted as to "create paychecks" for every company, which is typically not the case. Typically, the payroll department only "create paychecks" for the company to which the payroll department belongs. In some cases, some large companies have multiple payroll departments and each payroll department is only required to "create paychecks" for some specific departments. So, the goal "create paychecks" should be parameterized as "create paycheck for departments" rather than creating a set of similar goals such as "create paycheck for the sales department", "create paycheck for the marketing department", and "create paycheck for the financial department". In OMACS, the general goal "create paychecks" can be represent as g, and the parameterized goal "create paycheck for departments" can be represented as $g(departments)$. However, parameterized goals must be instantiated with concrete values before they can be achieved. So, for the example given above, the parameterized goal could be instantiated as "create paycheck for the sales department" ($g(sales)$).

1.1.3 Active Goal Set

Even though the organization is trying to achieve the set of leaf goals, the organization may not be trying to achieve all of the leaf goals at the same time for various reasons. Thus, OMACS introduces the notion of an active goal set ($\boldsymbol{G_A}$). The active goal set ($\boldsymbol{G_A} \subseteq \boldsymbol{G_L}$) represents the leaf goals that the organization is currently trying to achieve. As mentioned in § 1.1.2, if a goal is a parameterized goal, that goal must be instantiated with concrete values before the goal can be placed into the active goal set.

Using the payroll example, the parameterized goal "create paycheck for departments" cannot be in the active goal set unless the goal is instantiated with a department such as "create paycheck for the sales department", "create paycheck for the marketing department", and "create paycheck for the financial department". Doing

otherwise would cause confusion, the parameterized goal "create paycheck for departments" is too general. The payroll department would ask questions like "which department?". Thus, the parameterized goal "create paycheck for departments" must be instantiated.

Even with the instantiated parameterized goal "create paycheck for the sales department", the goal may still not be placed into the active goal set. Typically, paychecks are a time based event such as occurring on a bi-weekly or monthly basis. Thus, OMACS provides the notion of event-based triggered goals. A triggered goal (which can be either parameterized or non-parameterized) can only be placed into the active goal set when a specific event occurs while trying to achieve some other goal. Triggers are described in more detail in § 1.1.4.

OMACS also provides the notion of sequential achievement of goals. OMACS introduces the *precedes* relationship between goals to allow the sequential achievement of goals. Thus, if a goal is preceded by some other goal, the former goal cannot be placed into the active goal set until the latter goal has been achieved. The *precedes* relationship is be described in more detail in § 1.1.5.

According to the KAOS model, OR goals can be achieved with the achievement of only one of its subgoals. Thus, not all subgoals of an OR goal needs to be in the active goal set. In fact, if there are any OR goals in the goal tree then an organization only needs to achieve a subset of the leaf goals in order to achieve the top-level goal. This would potentially allow organizations to maximize their effectiveness in achieving the top-level goal.

Finally, the policies of an organization can further restrict which leaf goals from the active goal set are actually being assigned for achievement by agents. This set is known as the working goal set ($\boldsymbol{\omega_G}$), where $\boldsymbol{\omega_G} \subseteq \boldsymbol{G_A}$. In § 1.1.9, policies are described in more detail.

1.1.4 Triggers

The *triggers*[4] relation restricts goals from being placed into the active goal set until a specific event has occurred. Furthermore, when the specified event occurs, the triggered goal is placed into the active goal set. Following the example of the payroll department, lets say that another goal of the department is to "monitor all employees' work hours" and paychecks are issued every month. So, the parameterized goal "create paycheck for the sales department" is only triggered on a monthly basis by the goal "monitor all employees' work hours".

The *triggers* relation is inherited by subgoals. For instance, if g_1 (triggering goal) triggers g_2 (triggered goal) then all subgoals of g_1 must be able to trigger g_2. A high-level conceptual view of the goal tree allows the *triggers* relation between non-leaf goals. However, at the low-level, if there are any triggering goals that are non-leaf goals then these *triggers* relations are transformed into *triggers* relations of all the children of the triggering goals. This process is repeated until there are no more triggering goals that are non-leaf goals. This transformation is possible because triggers are inherited, only leaf goals can be in the active goal set, and events occur only during the achievement of a goal. Thus, triggering goals are limited to only leaf goals in OMACS. Figure 1.2 shows the transformation of the *triggers* relation. However, the transformation can only be applied to triggering goals and not triggered goals because of the ambiguity of OR goals that are triggered. For instance, g_1 triggers g_2, g_2 is an OR goal, and g_2 has two leaf goals. When g_2 is triggered, only one of the leaf goals is triggered. The procedure to determine which of the leaf goals is triggered is non-deterministic.

Typically, triggered goals are parameterized based on additional information provided by the event that triggered the goal. For example, the event that triggers the parameterized goal "create paycheck for the sales department" must have the name of the employees in the sales department.

[4]In the new version, the *triggers* relation exists in the goal model.

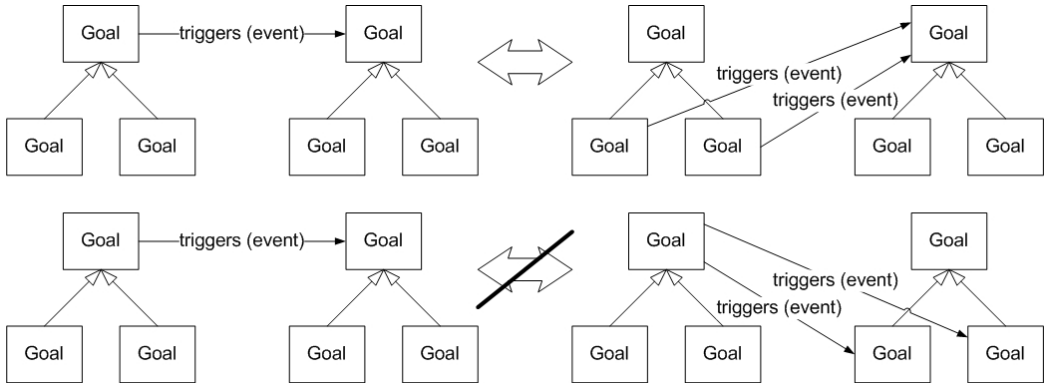


Figure 1.2: Triggers Transformation

1.1.5 Precedes

The *precedes*[5] relation allows the specification of sequential achievement of goals. For instance, if g_1 precedes g_2 then g_1 must be achieved before g_2 can be placed into the active goal set. Continuing with the payroll example, lets say that the parameterized goal "create paycheck for the sales department" has two subgoals: "get the name and the pay of an employee", and "print the paycheck". The goal "print the paycheck" should not be achieved without first knowing who and how much to pay the employee. To ensure that the goal "print the paycheck" is not achieved before the goal "get the name and the pay of an employee" is achieved, the *precedes* relation is used to specify that the goal "get the name and the pay of an employee" precedes the goal "print the paycheck".

The *precedes* relation is inherited by subgoals. For instance, if g_1 (preceding goal) precedes g_2 (preceded goal) then g_1 must precede g_2 and every subgoals of g_2. A high-level conceptual view of the goal tree allows the *precedes* relation between non-leaf goals. However, at the low-level, if there are any preceded goals that are non-leaf goals then these *precedes* relations are transformed into *precedes* relations of all the children of the preceded goals. This process is repeated until there are no more preceded goals that are non-leaf goals. This transformation is possible because precedes are inherited, and only leaf goals can be in the active goal set. Thus, OMACS

[5]In the new version, the *precedes* relation exists in the goal model.

limits the preceded goals to only leaf goals. Figure 1.3 shows the transformation of the *precedes* relation. However, the transformation can only be applied to preceded goals and not preceding goals because of the ambiguity of preceding OR goals. For instance, g_1 precedes g_2, g_1 is an OR goal, and g_1 has two leaf goals. g_1 is achieved when one of its leaf goals is achieved and g_2 can now be achieved. The process by which g_1 is achieved is non-deterministic.

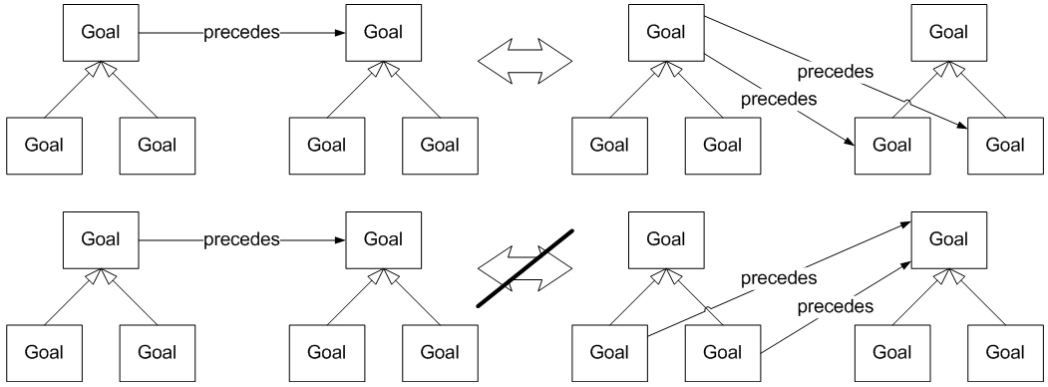


Figure 1.3: Precedes Transformation

1.1.6 Roles

Every organization has a set of roles that is required for an organization to achieve the set of goals of that organization. Generally, a role is capable of achieving multiple goals. However, the ability of that role to achieve those goals varies. Thus, OMACS defines a function `achieves()` to reflect the ability of a role to achieve a goal. The `achieves()` function returns a real value ranging from $0 \ldots 1$, where 0 indicates that a role is incapable of achieving the given goal and 1 indicating that a role is fully capable of achieving the given goal. Furthermore, since a goal can be achieved by multiple roles, the `achieves()` function allows the organization to pick the role that is most capable of achieving a particular goal.

In OMACS, roles are played by agents. However, there are requirements that agents must first fulfill before playing a given role. The requirements are represented in the form of capabilities available in the organization. Thus, roles require a certain

subset of capabilities and agents must possess the required subset in order to play a particular role.

However, possessing the required capabilities may not be a sufficient requirement for playing a role. Thus, OMACS introduces the function `rcf()`, also known as the role capability function. The `rcf()` returns a real value ranging from $0 \ldots 1$, where 0 indicates that the agent is incapable of playing the given role and 1 indicating the the agent is fully capable of playing the given role. Typically, capabilities do not contribute uniformly to the requirements of a role. Some capabilities might have more or less contribution than other capabilities. Thus, the `rcf()` allows roles to indicate the importance of capabilities when computing the result.

An example for using the `rcf()` is that there are two agents ($Agent1$ and $Agent2$), one role ($Role1$), and one capability ($Capability1$). Both agents possesses $Capability1$ but $Agent1$ has a score of 0.8 while $Agent2$ has a score of 0.4. $Role1$ requires $Capability1$. Without the `rcf()`, both agents are able to achieve $Role1$. However, with the `rcf()`, it is now possible to indicate that $Role1$ requires $Capability1$ with a score of at least 0.5. Thus, the `rcf()` when applied to $Agent1$ will return a non-zero score, if there are no other stipulations. However, even though $Agent2$ possesses $Capability1$, $Agent2$ lack the necessary expertise of $Capability1$ to achieve $Role1$. Thus, the `rcf()` when applied to $Agent2$ will return 0.

However, from the perspective of OMACS, the `rcf()` only indicates that $Agent2$ is unable to play $Role1$. OMACS is unable to find out that agents need to have a score of 0.5 or more of $Capability1$ to play $Role1$. In other words, the internal workings of the `rcf()` is hidden from OMACS.

1.1.7 Agents

Every organization has a set of heterogeneous agents. OMACS defines agents as "computational system instances that inhabit a complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals" [DeL05].

> Thus, agents are assumed to exhibit the attributes of autonomy, reactivity, pro-activity, and social ability. Autonomy is the ability of agents to control their actions and internal state. Reactivity is an agent's ability to perceive its environment and respond to changes in it, whereas pro-activeness ensures agents do not simply react to their environment, but that they are able to take the initiates in achieving their goals. Finally, social ability allows agents to interact with other agents, and possibly humans, either directly via communication or indirectly through the environment [DM05].

One of the reasons for the existence of organizations is to provide an effective mechanism for distributing goals to agents such that the top-level goal is achieved. Because of the heterogeneous nature of agents, there are some agents that are better at playing some roles than other agents. In addition, agents can be assigned to play more than one role, and roles can be assigned to more than one agent.

There are two reasons for the disparity: agents may possesses different sets of capabilities, and agents may have different expertise of the same capability. It is trivial to differentiate agents who have differing sets of capabilities. However, differentiating between agents who have the same set of capabilities is not so easy if the only information available is whether or not an agent possesses a capability. Thus, OMACS introduces the `possesses()` function. The `possesses()` function returns a real value ranging from $0 \ldots 1$, where 0 indicates that an agent does not possess the given capability and 1 indicates that an agent is fully competent in using the given capability. Thus, there can be two or more agents possessing the same set of capabilities but each agent could potentially have very different `rcf()` scores. Therefore, it is not sufficient to know that an agent possesses specific capabilities. The level of competence of agents in using those capabilities must also be known.

1.1.8 Capabilities

Capabilities are essential in determining what roles agents are capable of playing, as roles require capabilities and agents possess capabilities. Capabilities can represent a wide variety of abilities, both soft and hard. Some examples of soft abilities are having

access to resources, communications, and executing computational algorithms. Hard abilities typically model the abilities of robots such as sensors and effectors. Sensors allow the perception of the environment, and effectors allow interaction with the environment [DeL05].

1.1.9 Policies

Every organization has a set of policies. Policies provide additional constraints on the structure of organization such as modifying the relationships among agents, roles, capabilities, and goals. Policies also provide a mechanism for how organizations should act in certain situations. There are two types of policies in OMACS: assignment policies ($\boldsymbol{P_\Phi}$) and behavioral policies. However, only assignment policies are relevant within the scope of this thesis.

1.1.9.1 Assignment Policies

Assignment policies allow additional constraints on the assignment set. Without assignment policies, agents can be assigned to play a role if the `rcf()` returns a value greater than zero. For instance, there could be an assignment policy that says "agent x can only play role y" which effectively prevents 'agent x' from playing 'role z' even if the `rcf()` returns a value greater than zero.

1.1.9.2 Behavioral Policies

Behavioral policies constrain how the entities (agents, roles, and goals) of OMACS should interact in relation to one another. Behavioral policies deal with events and the reactions that should occur. [DeL02] talks about organizational rules (which are known as behavioral policies). For instance, in a conference paper review, if an author submits a paper then that paper should be reviewed by three reviewers[6].

[6]The example was taken from Dr. Scott A. DeLoach's working notes.

1.2 Scope and Objectives

The objective of this thesis is to investigate if an efficient generic reorganization algorithm can be developed and if not to investigate some basic characteristics of OMACS that can lead to a more efficient time complexity. As a result, this thesis introduces a reorganization algorithm that finds an optimal organization score. This algorithm is designed such that a distributed variant can be easily implemented. In this thesis, a simple distributed version of the algorithm is provided. Also, an analysis of the time complexity of the algorithm and some test results on an implementation of the algorithm is provided. From a combination of the analysis and the test results, certain characteristics that can be used with OMACS for designing "good" and "efficient" models is shown.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides background information on the different types of search algorithms currently available.

Chapter 3 describes two versions of the algorithm developed for OMACS. The centralized version is described first followed by the simple distributed version.

Chapter 4 analyzes the time complexity of the two versions of the algorithm. In addition, a high-level analysis of message complexity for the simple distributed version of the algorithm is provided.

Chapter 5 presents an implementation of the two versions of algorithm in a high-level simulator that uses the current implementation of OMACS.

Chapter 6 evaluates the implementation of the centralized version of the algorithm through a series of test cases.

Chapter 7 summarizes the contributions of this thesis and provides suggestions for future improvements.

Chapter 2

Background

This chapter provides background information on a number of related areas that lead to the work in this thesis. § 2.1 provides background information on related work. § 2.2 looks at some related reorganization algorithms applied to a specific problem. And § 2.3 looks at some strategies for search algorithms.

2.1 Related Work

Many multiagent systems exhibit characteristics of an organization, whether it is implicit or by design [CG99]. According to Carley and Gasser, organizations are "heterogeneous, complex, dynamic nonlinear adaptive and evolving systems" [CG99] and because of this, organizations are hard to analyze. To allow analysis of organizations, they introduced the notion of Computational Organization Theory (COT). COT models formalizes various organizational concepts in attempts to study and analyze the attributes or emergent properties of organizations.

One of the first work that formalized some of these organization concepts is the Aalaadin model [FG98]. The Aalaadin model formalized the organization concepts of agents, groups, and roles. These formalized concepts are now used in many other organization-based models. In Aalaadin, an agent is defined as an entity that plays roles, a group is defined as a set of agents, and a role is defined as the function of agents within a group. Later, the Aalaadin model was extended to the AGR model [FGM03]. The formalized concepts of agents and roles are used in OMACS.

Kirn and Gasser [KG98] provided a specification for organizational design by formalizing the concept of coordination. They defined the term "organizational position" as "a set of role expectations", which can be be formally specified by their Vienna Development Method (VDM) notation. The concept of coordination was present in earlier versions of OMACS but was later removed because coordination does not contribute to the process of reorganization.

Ferber, Gutknecht, Jonker, Müller, and Treur [FGJ+02] formalized the behavioral aspects of organizations through the use of requirements engineering. Their approach uses the definitions from the AALAADIN model, in which four types of behavioral requirements are identified: single role behavior requirements, intragroup interaction requirements, intragroup communication successfulness requirements, and intergroup interaction requirements. The concept of behavior is present in OMACS in the form of behavioral policies.

Odell, Nodine, and Levy [ONL05] introduced a metamodel for modeling multiagent systems in an extension of Unified Modeling Language (UML). The concepts used in their metamodel follows from the AALAADIN model of agents, groups, and roles. OMACS is a metamodel in addition to being a COT model.

The research work listed above highlights some of the work done in formalizing organizational theory. These formalizations in turn helped in the development of more complete organization-based models. And, so the next two subsections looks at two complete organization-based COT models: OperA (§ 2.1.1) and OMNI (§ 2.1.2).

2.1.1 OperA

OperA (*Organizations per Agents*) [Dig04] is a multiagent systems framework that allows specification of organizational objectives and structure by allowing autonomous and heterogeneous agents to enter and leave the organization. OperA framework allow these agents to coordinate and collaborate with other agents in the organization to achieve the organizational objectives. To achieve the coordination and collaboration, OperA provides the notion of abstract protocols that allows agents to interact with

other agents regardless of how the agent was designed. Thus, the OperA framework consists of three parts: Organizational Model, Social Model, and Interaction Model.

The Organizational Model defines the characteristics of the organization using the following four structures: social, interaction, normative, and communicative. The social structure defines the objectives of the organization, the roles that exists in the organization, and the type of coordination model used by the organization. The interaction structure defines interaction among agents as scene scripts, which is played out by the roles from the organization. In addition, the interaction structure introduces a partial ordering to the scene scripts, which provides information about the interaction that occurs among roles. The normative structure defines the norms and regulations [Dav01] of the organization, which are split into three categories: role norms, scene norms, and transition norms. Role norms defines how agents playing a role should act. Scene norms defines how roles in scenes should act. And transition norms defines how roles are able to move from one scene to another. The communicative structure defines the ontology that is used by the organization for communication, which consists of the domain knowledge (knowledge representation language) and the agent interaction (communication language).

In the Social Model, social contracts constrains the actions of agents in the organization. A social contract defines conditions that an agent must meet before assuming a role in the contract as well as rules that the agent must follow after assuming the role. An agent playing a role in a social contract is termed a role-enacting agent.

The Interaction Model allows the dynamic enactment of scripts (defined in the Organizational Model) by role-enacting agents (defined in the Social Model). Because scripts from the Organizational Model are abstract, role-enacting agents in a script must first agree on the interaction protocol that will be used for the script. Thus, the interaction contract is formed.

Although OperA provides the means to model an organization by describing the interactions that occur among the agents, the overall objectives of the organization are encapsulated within the roles specified in the social structure of the Organization

Model. Basically, the goals have been merged with the roles. In OMACS, the goals and roles are separated. This allows for a stronger expression of the relationship between roles and goals. However, OperA provides a more complete framework from design to implementation. Currently, OMACS is lacking the implementation area.

2.1.2 OMNI

OMNI (*Organizational Model for Normative Institutions*) [DVSD04] is a framework designed for closed and open multiagent systems. OMNI allows a wide range of multiagent systems to be modeled. OMNI caters to open multiagent systems that is able to allow heterogeneous agents to enter the organization with their own goals, beliefs, and capabilities. Furthermore, OMNI does not assume that agents that enter the organization are cooperative agents.

OMNI is based on two models: OperA [Dig04] and HarmonIA [VSD03]. The OMNI framework consists of three dimensions. The three dimensions are the Normative Dimension, the Organizational Dimension, and the Ontological Dimension. Furthermore, each of the three dimension is broken up into three levels: Abstract Level, Concrete Level, and Implementation Level.

The Abstract Level defines the main objectives (*statutes*) of the organization, which are the overall goals of the organization. The manner in which the organization achieves the objectives are guided by the *values* of the organization. The *values* of the organization are beliefs that defines what is good or bad. Lastly, the environment in which the organization functions in is defined by the *context*. The Concrete Level refines the definitions of the Abstract Level further by defining the norms [DD01] [EPS01] and rules [Zam02] of the organization, the roles in the organization, landmarks, and concrete ontological concepts. And finally, the Implementation Level implements the definitions from the Concrete Level. Examples of the mechanisms are mechanisms for playing roles and mechanisms for enforcing norms and rules.

The purpose of the Normative Dimension is to verify that the organization follows norms and rules defined by the organization. In the Abstract Level of the Normative

Dimension, the *values* are the norms which are used to define concepts that determine the usefulness of a situation. The Concrete Level refines the norms from the Abstract Level further resulting in what is called the Norm Level. The norms from the Norm Level is then used to create the rules, violations, and sanctions, which is known as the Rule Level. The norms and rules from the Concrete Level are implemented in the Implementation Level using either of the two approaches: "creating a rule interpreter that any agent entering the organization will incorporate" and "translating the rules into protocols to be included in the interaction contracts".

The Abstract Level of Organizational Dimension defines the organization's objectives. These objectives are only objectives that can be observed from outside the organization. Internal or unobservable objectives are not defined here. The Concrete Level defines how these objectives can be achieved by providing two structures: social structure and interaction structure. The social structure defines the roles of the organization and the interaction structure defines "a set of meaningful scenes". The two structures are implemented in the Implementation Level. The social structure results in the social model and the interaction structure results in the interaction model.

The Ontological Dimension defines mechanisms for coordination and collaboration within the organization. The Abstract Level provides a meta-ontology that defines all the concepts of OMNI such as the norms and rules, the roles and groups, the violations, the sanctions, and the landmarks. The Concrete Level are split into two parts: the Concrete Domain Ontology and the Procedural Domain Ontology. The Concrete Domain Ontology "includes all the predicates and elements that appear during the design of the Organizational and Normative Structure". The Procedural Domain Ontology provides terms that will be used in the Implementation Level.

OMNI provides a formal logical semantics for every part of the framework. This allows OMNI models to be verified for correctness and consistency. Currently, OMACS lacks a formal logical semantics. However, OMACS is in the process of being formalized.

2.2 Reorganization Algorithms

Generally, there is no single organization that is sufficient for any situation [IGY92] [HL05] [CG99]. Therefore, there is a need to change the structure of the organization (reorganization) so that the new organization is "better" when adapting to changes. "Better" is a subjective term that can mean more efficient or more flexible. Often, efficiency and flexibility are opposing forces [RDK06]. Increasing efficiency may decrease flexibility and vice versa. According to [DDS04], two types of reorganization can occur: behavioral change and structural change. Behavioral change occurs when a new agent joins the multiagent system, when an agent leaves the multiagent system, and when agents agree on some interation protocol. Structural change occurs when the interaction between agents changes, and when environmental changes requires modifications to the design model. The organization models mentioned in this section are designed for a specific problem domain.

Ishida, Gasser, and Yokoo [IGY92] introduced the concept of reorganization to a production system, in what they call Organization Self-Design (OSD). OSD consists of two new organization primitives: decomposition and composition. Decomposition divides one agent into two, while composition combines two agents into one. Their OSD approach is able to adapt to a particular set of environmental changes such as changes in response time requirements, changes in number of requests per time unit, and changes in demand for resources. Their experimental results show increased performance when OSD is used.

Barber and Martin [BM01] provided a framework for multiagent systems called Adaptive Decision-Making Frameworks (ADMF). In their paper, ADMF assumes that resources that the system requires do not change, the capabilities of agents do not change, and tasks responsibilities do not change. The focus of reorganization is primarily on two types of relationships among agents: "decision-making control" and "authority-over". Agents designated as "decision-making control" agents are agents that decides how goals are achieved. Agents designated as "authority-over" agents are agents that are under the command of "decision-making control" agents. Their

experiments show promising results of improved performance when compared to a static model.

In [ZCLL04], the authors proposed a multiagent system framework for a distributed information retrieval system for use in a peer-to-peer network. Their implementation relies on two algorithms: the agent-view algorithm, and the distributed search algorithm. Their focus is primarily on the Agent-View Reorganization Algorithm (AVRA). The AVRA creates a topology consisting of agents that are grouped into non-disjoint clusters. A cluster consists of agents that knows the same information. Results from their experiments show significant performance increase for information retrieval.

The algorithms listed above are tied to a specific problem domain. There may not be performance gains when applied to a different problem domain. Thus, the next section looks at general search strategies.

2.3 Search Strategies

There are two broad areas of search strategies: uninformed search and informed search. Algorithms in these areas are evaluated in the following four categories: completeness, time complexity, space complexity, and optimality. In order for an algorithm to be complete, that algorithm is guaranteed to find a solution if one exists. Time complexity means how long the algorithm takes to find a solution. Space complexity means how much memory is required for an algorithm to perform the search. An optimal algorithm returns the best solution if there is more than one solution. [RN95]

2.3.1 Uninformed Search

An uninformed search algorithm can be applied to a wide variety of problems because uninformed search algorithm does not rely on utilizing information about the problem domain. Because uninformed search algorithms do not rely on information about the problem domain, a general implementation of uninformed search algorithms can

be applied to a wide variety of problems. However, the search space required by uninformed search algorithms are exceedingly large for small examples.

Two popular uninformed search algorithms are looked at: depth-first and breath-first. The two uninformed search algorithms are not the only algorithms available; there are more uninformed search algorithms such as uniform-cost search and iterative deepening search.

2.3.1.1 Depth-First Search

Depth-first search [RN95] works in a similar way to a LIFO (last-in, first-out) queue. An initial node is select and inserted into the queue. The following steps then occurs for the depth-first search algorithm:

1. If the queue is empty then the depth-first search algorithm stops because there is no path from the initial node to the goal node. Otherwise, the depth-first search algorithm removes the last node (n) from the queue.

2. If the removed node (n) is a goal node then the depth-first search algorithm returns the path taken from the initial node to reach n and stops. Otherwise, n is expanded to obtain a list (l) of all adjacent nodes that can be reached from n and n is then inserted into the "visited" list.

3. Every node l_i from l is checked against the "visited" list. If $l_i \notin$ "visited" list then l_i is inserted into the queue.

4. Repeat step 1.

The above description assumes that all edges have a uniform "cost". If the "cost" is different for edges, then step 3 will have to be modified appropriately to handle the non-uniform "cost" of edges. Figure 2.1 shows an example of the order in which the nodes are visited.

The following is the evaluation of the depth-first search on the four categories:

- Not complete because depth-first search may get stuck in an infinite path.

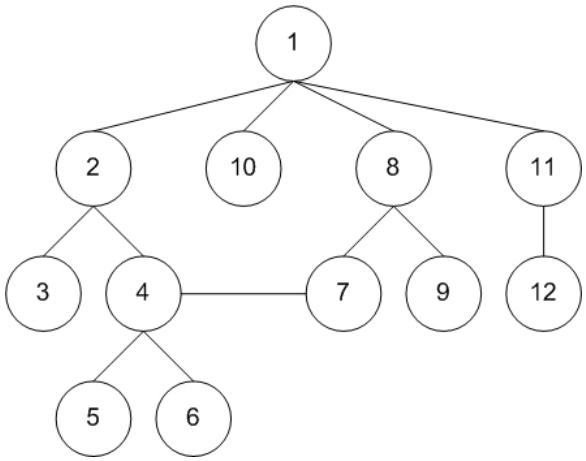


Figure 2.1: Depth-First Search Example

- Not optimal because depth-first search may find a solution with a bigger depth when a solution with a smaller exists.
- Time complexity is $O(b^d)$, where b is the number of adjacent nodes that each node has and d is the depth of the solution.
- Space complexity is $O(bm)$, where b is the number of adjacent nodes that each node has and m is the maximum depth of the search space.

In OMACS, there is no trivial mechanism to transform the problem of reorganization into a graph where the depth-first search algorithm can be applied. Two of the transformation problems are: a structured representation of the reorganization problem, and how the "cost" of edges are computed.

2.3.1.2 Breath-First Search

Breath-first search [RN95] works in a similar way to a FIFO (first-in, first-out) queue. An initial node is selected and inserted into the queue. The following steps then occurs for the depth-first search algorithm:

1. If the queue is empty then the breath-first search algorithm stops because there is no path from the initial node to the goal node. Otherwise, the breath-first search algorithm removes the first node (n) from the queue.

2. If the removed node (n) is a goal node then the breath-first search algorithm returns the path taken from the initial node to reach n and stops. Otherwise, n is expanded to obtain a list (l) of all adjacent nodes that can be reached from n and n is then inserted into the "visited" list.

3. Every node l_i from l is checked against the "visited" list. If $l_i \notin$ "visited" list then l_i is inserted into the queue.

4. Repeat step 1.

The above description assumes that all edges have a "cost" that is obtained from a function that is non-decreasing based on the depth of the node. However, if the "cost" is different for edges, then step 3 will have to be modified appropriately to handle the non-uniform "cost" of edges. Figure 2.2 shows an example of the order in which nodes are visited.

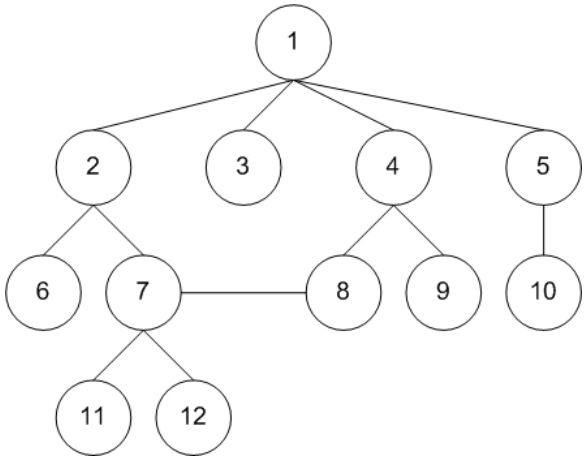


Figure 2.2: Breath-First Search Example

The following is the evaluation of breath-first search on the four categories:

- Complete because breath-first search will find a solution if one exists.

- Optimal if "costs" are uniform. Otherwise, breath-first search finds the solution with the shortest path.

- Time complexity is $O(b^d)$, where b is the number of adjacent nodes that each node has and d is the depth of the solution.
- Space complexity is $O(b^d)$, where b is the number of adjacent nodes that each node has and d is the depth of the solution.

The memory requirements of the breath-first search algorithm is unacceptably large. Even though the time and space complexity are the same, the memory requirements poses a significant problem because memory is limited, whereas time is not. If a problem requires 10 gigabytes of memory and the computer does not have that memory, it is not possible to perform the search without upgrading the computer. However, if there is sufficient memory, and the solution requires 10 months of running time, it is possible to leave the computer running for 10 months to obtain the solution.

2.3.2 Informed Search

An informed search algorithm relies on heuristics, which incorporates information about the problem domain, to decide which search path to explore. A good heuristic will enable an informed search algorithm to perform more efficiently than an uninformed search algorithm. In uninformed search algorithms, the search space can be exceedingly large and sometimes it is just not feasible to search the whole search space. That is the purpose of heuristics, heuristics should allow for a more focused search in the search space that seems to lead closer to the goal. However, there are some cases where the best solution involves taking some really convoluted path to reach the goal.

The following sections look at two of the more popular informed search algorithms: best-first search and A* search. [RN95]

2.3.2.1 Best-First Search

Best-first search [RN95] is an optimization of the depth-first search algorithm. Best-first search utilizes an evaluation function to estimate the score of a node. Instead of a LIFO queue, best-first search uses a priority queue. Nodes in the queue are sorted

by their score. The node that is removed from the queue is the one with the best score (usually the node with the lowest "cost"). The simplest form of the best-first search is also known as the greedy search, which uses the evaluation function $h(n) =$ estimated "cost" from current node to the goal node. An initial node is select and inserted into the queue. The following steps then occurs for the depth-first search algorithm:

1. If the queue is empty then the best-first search algorithm stops because there is no path from the initial node to the goal node. Otherwise, the best-first search algorithm removes the best node (n,$h(n)$) from the queue.

2. If the removed node (n,$h(n)$) is a goal node then the best-first search algorithm returns the path taken from the initial node to reach n and stops. Otherwise, n is expanded to obtain a list (l) of all adjacent nodes that can be reached from n and (n,$h(n)$) is then inserted into the "visited" list.

3. Every node l_i from l is checked against the "visited" list. If $l_i \notin$ "visited" list then the evaluation function is applied to l_i to obtain (l_i,$h(l_i)$) and inserted into the queue.

4. Repeat step 1.

The following is the evaluation of best-first search on the four categories:

- Complete if depth of search space is finite.

- Not optimal because the best-first search may find a solution with a total "cost" that is higher than then optimal one.

- Time complexity is $O(b^m)$, where b is the number of adjacent nodes and m is the maximum depth of the search space.

- Space complexity is $O(b^m)$, where b is the number of adjacent nodes and m is the maximum depth of the search space.

Best-first search presents the same problems as breath-first search where the memory requirement is unacceptably high.

2.3.2.2 A* Search

A* search [RN95] is a refinement of the best-first search algorithm. A* search performs the search by taking in account the path "cost". The path "cost" is the sum of the "costs" taken from the initial node to the current node. A* search uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the path "cost" and $h(n)$ is the estimated "cost" from current node to the goal node. The following steps occurs for the A* search algorithm:

1. If the queue is empty then the A* search algorithm stops because there is no path from the initial node to the goal node. Otherwise, the A* search algorithm removes the best node $(n,f(n))$ from the queue.

2. If the removed node $(n,f(n))$ is a goal node then the A* search algorithm returns the path taken from the initial node to reach n and stops. Otherwise, n is expanded to obtain a list (l) of all adjacent nodes that can be reached from n and n is then inserted into the "visited" list.

3. The evaluation function is applied to every node l_i from l to obtain $(l_i,f(l_i))$.

4. $(l_i,f(l_i))$ is checked against the "visited" list. If $l_i \notin$ "visited" list then insert $(l_i,f(l_i))$ into the "visited" list. If $l_i \in$ "visited" list as $(l_i,f(l_k))$ then if $f(l_i) < f(l_k)$ then replace $(l_i,f(l_k))$ with $(l_i,f(l_i))$.

5. Repeat step 1.

The following are some points to note about the A* search algorithm:

- "Complete on locally finite graphs (graphs with a finite branching factor) provided there is some positive constant δ such that every operator costs at least δ" [RN95].

- Optimal if the heuristic is admissible. An admissible heuristic is where the $f(n)$ never overestimates the cost of reaching the goal node. In other words, a heuristic is admissible when $h(n) \leq h^*(n)$ for all n, where $h^*(n)$ is the actual cost of reaching the goal node from n. An example of a admissible heuristic is $h(n) = 0$ for all n.
- Time complexity
- Space complexity

Since A* search is a refinement of the best-first search algorithm, A* search also suffers from an unacceptable memory requirement.

2.4 Summary

In summary, this chapter looks at early work that formalized some aspect of organization theory, which eventually lead to the development of the two organization-based COT models described in § 2.1.1 and § 2.1.2. Following that, the next section looks at three reorganization algorithms that is designed for a specific problem domain, and how their performance benefits might not transfer when applied to other domains. And lastly, four types of general search algorithms are discussed. The main limitation of adopting any one of the algorithms is with the transformation to a structured graph-like representation. The next chapter looks at an algorithm developed for OMACS that resulted from the study of the domain-specific reorganization algorithms and the four general search algorithms.

Chapter 3

Reorganization Algorithm

In this chapter, the two versions of the reorganization algorithm developed specifically for OMACS are described. The algorithm is guaranteed to find the optimal organizational score, if one exists. The optimal organizational score is the highest organizational score returned by the `oaf()` for differing assignment sets. Furthermore, the reorganization algorithm can be used to determine the initial organization as well.

The centralized version of the algorithm is described first. The centralized version relies on a "master" agent to compute the optimal organizational score using a given `oaf()`. However, since the internal workings of the `oaf()` is not exposed to the algorithm and that there are no specifications to follow for constructing an `oaf()`, the only way to compute the optimal organization score is through a brute force search. However, some shortcuts are used so that paths that are known to be invalid before the algorithm starts are not explored. One of these shortcuts is using the relationship between goals and roles. Thus, there is no need to test every goal with every role based on the `achieves()` function. The set of roles can be directly extracted from the goals by using that relationship. For the purposes of this thesis, the centralized version of the algorithm is known as the *Centralized Brute Force* (*CBF*) algorithm.

Following the description of the *CBF* is the description of the simple distributed version of the algorithm. The idea is that every agent in the organization contributes computing power to compute the optimal organization score. The reason for using the distributed approach is so that the time complexity of the *CBF* can be be reduced.

For the purposes of this thesis, the distributed version of the algorithm is known as the *Distributed Brute Force* (*DBF*) algorithm.

Before the description of the two versions of the algorithm, the following are the underlying assumptions made by the algorithm.

- Every agent within the organization is able to communicate with each other, either directly or indirectly. Agents that are unable to communicate are not considered to be part of the organization at the time the algorithm starts.
- Every organization provided to the algorithm is valid, particularly the `rcf()` and the `oaf()` are deterministic.

Furthermore, an example organization is used throughout this chapter to better explain the algorithm. The example organization consists of three working agents ($\boldsymbol{\omega_A}_1$, $\boldsymbol{\omega_A}_2$, and $\boldsymbol{\omega_A}_3$), three roles ($\boldsymbol{R}_1$, $\boldsymbol{R}_2$, and $\boldsymbol{R}_3$), and three working goals ($\boldsymbol{\omega_G}_1$, $\boldsymbol{\omega_G}_2$, and $\boldsymbol{\omega_G}_3$).

The algorithm will find the optimal organization score based on a given set of leaf goals ($\boldsymbol{\omega_G} \subset \boldsymbol{G_A}$) and a given set of agents ($\boldsymbol{\omega_A} \subset \boldsymbol{A}$). Thus, the algorithm is able to find the optimal organization score on a portion of the organization. In order to find the optimal organization score for the complete organization, the given inputs should be $\boldsymbol{\omega_G} = \boldsymbol{G_A}$ and $\boldsymbol{\omega_A} = \boldsymbol{A}$.

3.1 Centralized Brute Force Version

This section looks at how the *CBF* version of the algorithm works. Furthermore, the *CBF* serves as the basis for comparison by the *DBF*. The pseudo code for the *CBF* is given in Figure 3.1. The *CBF* assumes that there is a "master" agent in the organization with the complete knowledge of the organization and that this "master" agent is the agent responsible for reorganizing. This agent is simply known as $\boldsymbol{A}_0$.

The *CBF* computes the optimal organizational score in two phases.

1. Creation of the data-structure necessary for computing the optimal organizational score.

```
function CBF(Organization, ω_G, ω_A)
  maps ← {}
  for each ω_Gg do
    for each R_r that achieves ω_Gg do
      maps ∪ ⟨R_r, ω_Gg⟩
    end for
  end for
  λ ← reduce(powerset(maps))
  links ← {}
  for each ω_Aa do
    for each λ_m do
      if ω_Aa is capable of λ_m then
        links ∪ ⟨ω_Aa, λ_m⟩
      end if
    end for
  end for
  combinations ← combination(reduce(links))
  for each combination_i from combinations do
    Φ ← {}
    for each link_l from links do
      Φ ∪ ⟨link_l.agent, link_l.set_combination_i⟩
    end for
    if P_Φ(Φ) is valid then
      if (score ← oaf(Φ)) > best.score then
        best ← ⟨score, Φ⟩
      end if
    end if
  end for
  return best.Φ
```

Figure 3.1: Centralized Brute Force Pseudo Code

2. Computing the optimal organizational score.

The disadvantage of creating the data-structures first and then computing the optimal organization score is that the space complexity is higher than other centralized variants that creates the data-structures on the fly. However, this approach allows the *CBF* to be easily distributed.

First, $\boldsymbol{A}_0$ creates all the valid mappings between roles and goals in the organization. In the algorithm, the $\boldsymbol{\omega_G}$ is used to create the mappings. The mappings $\langle \boldsymbol{R}_r,$

$\boldsymbol{\omega_{Gg}}\rangle$ are created by going through each working goal ($\boldsymbol{\omega_{Gg}}$) and mapping $\boldsymbol{\omega_{Gg}}$ with every role ($\boldsymbol{R}_r$) than can achieve $\boldsymbol{\omega_{Gg}}$. No checks are required to determine if $\boldsymbol{R}_r$ achieves $\boldsymbol{\omega_{Gg}}$ because the set of roles that achieves $\boldsymbol{\omega_{Gg}}$ can be obtained by using the relationship "achieved by", which returns a set of roles that achieves $\boldsymbol{\omega_{Gg}}$.

Once all the mappings are obtained, a power set of the mappings is created. This power set represents all the possible role-goal assignments that agents can be assigned to, regardless if an individual agent is actually capable of playing the role. Figure 3.2 shows an example power set of three mappings ($\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle$, $\langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle$, and $\langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle$), where each set from the power set is a possible mapping set ($\boldsymbol{\lambda}_m$).

$\{\}$
$\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle\}$
$\{\langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle\}$
$\{\langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$
$\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle\}$
$\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$
$\{\langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$
$\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$

Figure 3.2: Example Power Set

The size of the power set can be reduced by applying assignment policies ($\boldsymbol{P_\Phi}$) of the organization. Mapping sets from the power set can be removed if there is a policy that would indicate that a particular mapping set is invalid. For example, if there is a policy that says that an agent cannot be assigned to two or more roles, then the mapping sets $\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle\}$, $\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$, $\{\langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$, and $\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$ from Figure 3.2 can be removed from the power set. This would effectively reduce the size of the example power set from Figure 3.2 from 8 mapping sets to 4 mapping sets.

With this reduced power set, all agents from $\boldsymbol{\omega_A}$ are checked against each mapping set. If an agent ($\boldsymbol{\omega_{A}}a$) is capable of playing a mapping set ($\boldsymbol{\lambda}_m$), a link is created between $\boldsymbol{\omega_{A}}a$ and $\boldsymbol{\lambda}_m$. Figure 3.3 shows an example of the conceptual view of a links data structure of the three agents ($\boldsymbol{\omega_{A}}1$, $\boldsymbol{\omega_{A}}2$, and $\boldsymbol{\omega_{A}}3$) and the example power set

from Figure 3.2.

$\boldsymbol{\omega_{A1}} = \{\{\}, \{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}, \{\langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}\}$
$\boldsymbol{\omega_{A2}} = \{\{\}, \{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle\}, \{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle\}\}$
$\boldsymbol{\omega_{A3}} = \{\{\}, \{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle\}, \{\langle \boldsymbol{R}_2, \boldsymbol{\omega_{G2}}\rangle\}, \{\langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}\}$

Figure 3.3: Example Links Data Structure

Similarly, the number of links can be reduced by assignment policies in the organization. For example, if there is a policy that says that $\boldsymbol{\omega_{A3}}$ cannot play $\boldsymbol{R}_3$, then the link to the mapping set $\{\langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$ can be removed from $\boldsymbol{\omega_{A3}}$.

The reduced links data-structure is used for the second phase of the algorithm: computing the optimal organizational score. To compute an organizational score, an assignment set ($\boldsymbol{\Phi}$) is required. An assignment set is when all agents from $\boldsymbol{\omega_A}$ have been assigned to exactly one mapping set from their links data-structure. For instance, an assignment set could be $\boldsymbol{\omega_{A1}}$ is assigned to $\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega_{G3}}\rangle\}$, $\boldsymbol{\omega_{A2}}$ is assigned to $\{\}$, and $\boldsymbol{\omega_{A3}}$ is assigned to $\{\langle \boldsymbol{R}_1, \boldsymbol{\omega_{G1}}\rangle\}$. At this point, an organizational score can be computed.

Therefore, finding the optimal organizational score requires going through every possible combination of the assignment set and computing the organizational score for each combination, while keeping track of the highest organizational score. However, not all combinations are valid because there could be a policy that disallows some combinations. For example, there could be a policy that says that if $\boldsymbol{\omega_{A1}}$ plays $\boldsymbol{R}_1$, $\boldsymbol{\omega_{A3}}$ cannot play $\boldsymbol{R}_3$. Once the algorithm finishes going through all the combinations, the algorithm will have found the optimal organizational score and the assignment set that produced that score assuming one exists.

3.2 Distributed Brute Force Version

This section looks at a way to distribute the *CBF*. The approach used is very simple and the premise adopted is to have as little communications as possible among the agents. The *DBF* attempts to alleviate some of the computational complexity of the

CBF by parallelizing one aspect of the *CBF*: the generation of the data-structures before computing the optimal organizational score. The pseudo code for the *DBF* is given in Figure 3.4. The *DBF* relies on the assumption that every agent already has the complete knowledge of the organization. Or that the agents are able to obtain that knowledge before the algorithm begins.

```
function DBF(Organization, ω_G)
  maps ← {}
  for each ω_Gg do
    for each R_r that achieves ω_Gg do
      if ω_Aself is capable of R_r then
        maps ∪ ⟨R_r, ω_Gg⟩
      end if
    end for
  end for
  λ_self ← reduce(powerset(maps))
  send λ_self to all agents
  receive λ_i from all agents
  combinations ← combination(λ)
  for each combination_i from combinations do
    Φ ← {}
    for each λ_j from λ do
      Φ ∪ λ_{j_combination_i}
    end for
    if P_Φ(Φ) is valid then
      if (score ← oaf(Φ)) > best.score then
        best ← ⟨score, Φ⟩
      end if
    end if
  end for
  return best.Φ
```

Figure 3.4: Distributed Brute Force Pseudo Code

When the algorithm starts, each agent ($\boldsymbol{\omega}_{\boldsymbol{A}self}$) begins by computing their list of possible mappings $\langle \boldsymbol{R}_r, \boldsymbol{\omega}_{\boldsymbol{G}g} \rangle$. Creating a mapping is almost similar to the way the *CBF* creates the mappings with the exception that the mappings are specific to $\boldsymbol{\omega}_{\boldsymbol{A}self}$. A mapping is created for $\boldsymbol{\omega}_{\boldsymbol{A}self}$ by going through each working goal ($\boldsymbol{\omega}_{\boldsymbol{G}g}$) and mapping $\boldsymbol{\omega}_{\boldsymbol{G}g}$ with every role ($\boldsymbol{R}_r$) than can achieve $\boldsymbol{\omega}_{\boldsymbol{G}g}$ and $\boldsymbol{\omega}_{\boldsymbol{A}self}$ is capable

of playing $\boldsymbol{R}_r$. Figure 3.5 shows an example of the mappings for $\boldsymbol{\omega}_{\boldsymbol{A}self}$ using the example organization.

$\boldsymbol{\omega}_{\boldsymbol{A}1} = \langle \boldsymbol{R}_1, \boldsymbol{\omega}_{\boldsymbol{G}1} \rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega}_{\boldsymbol{G}3} \rangle$
$\boldsymbol{\omega}_{\boldsymbol{A}2} = \langle \boldsymbol{R}_1, \boldsymbol{\omega}_{\boldsymbol{G}1} \rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega}_{\boldsymbol{G}2} \rangle$
$\boldsymbol{\omega}_{\boldsymbol{A}3} = \langle \boldsymbol{R}_1, \boldsymbol{\omega}_{\boldsymbol{G}1} \rangle, \langle \boldsymbol{R}_2, \boldsymbol{\omega}_{\boldsymbol{G}2} \rangle, \langle \boldsymbol{R}_3, \boldsymbol{\omega}_{\boldsymbol{G}3} \rangle$

Figure 3.5: Working Agents Mappings

Once $\boldsymbol{\omega}_{\boldsymbol{A}self}$ has created all the mappings, a power set of mappings is created. The difference between the power set from the *DBF* and the *CBF* is that the power set from the *DBF* is specific to $\boldsymbol{\omega}_{\boldsymbol{A}self}$. Similarly, just like the *CBF*, the power set can be reduced in size by assignment policies of the organization.

With the reduced power set, $\boldsymbol{\omega}_{\boldsymbol{A}self}$ sends the reduced power set to every other agent. After which, $\boldsymbol{\omega}_{\boldsymbol{A}self}$ waits to receive the reduced power set from every other agent. No assumptions are made on how the reduced power set is sent. The reduced power set can be sent in some optimized form such that when $\boldsymbol{\omega}_{\boldsymbol{A}self}$ receives a reduced power set, there are no two or more instances of duplicate mappings.

When $\boldsymbol{\omega}_{\boldsymbol{A}self}$ receives all the reduced power sets from every other agent, $\boldsymbol{\omega}_{\boldsymbol{A}self}$ begins computing the optimal organizational score. The procedure for computing the optimal organization score in the *DBF* is exactly the same as the *CBF*. In § 7.2, other ways of designing a distributed algorithm are mentioned. However, the details of designs and implementation are left to future work.

3.3 Summary

In summary, this chapter describes the internal workings of the two versions of the algorithm. The *CBF* version finds the optimal organizational score by relying on a "master" agent to do all the computation. The *DBF* version distributes the work of generating the data-structures to all agents involved but computes the optimal organizational score individually. In the next chapter, the complexity of the algorithm is analyzed, particularly focusing on the time complexity. Also, space complexity

is analyzed when required and a high-level analysis of the message complexity is provided for future distributed algorithms to consider.

Chapter 4

Complexity Analysis

This chapter analyzes the complexity of the two versions of the algorithm: *CBF* and *DBF*. The focus is primarily on the *CBF* version since the *DBF* version is more of an illustrative example of how to distribute the algorithm.

In the analysis, time complexity of the algorithm is the primary concern. However, because some parts of the time complexity analysis rely on knowing the space complexity, the space complexity analysis is provided on a "need to" basis.

The following are the assumptions for the analysis of the algorithm.

- Instructions are executed in constant time.
- There are no assignment policies ($\boldsymbol{P_\Phi}$). Assignment policies have varying effects in reducing the size of data-structures. Some can reduce the size by an exponential factor, some a polynomial factor, and some a constant factor. For the purposes of analyzing the algorithm, there are no assignment policies. Further discussions on the effects of assignment policies are in § 6.3.2.
- The given organization is valid, particularly the `oaf()` and `rcf()` are deterministic. Deterministic means that if the same input is given to the `oaf()`, the `oaf()` will always return the same value. Similarly, the `rcf()` returns the same value for the same input. Furthermore, the set of working agents and the set of working goals all exists within the organization.

- The given organization is achievable. For an organization to be achievable, roles have to be achieved by at least one agent and goals have to be achieved by at least one role.

- Agents communicate via messages which are sent point to point. There are no mechanisms for broadcasting messages. However, a broadcast can be simulated by sending the same message to every agent.

With the stated the underlying assumptions, the following terms are defined for use in the analysis of the algorithm.

Definition 1 *let g be the cardinality of $\boldsymbol{\omega_G}$ ($g = |\boldsymbol{\omega_G}|$), a part of validity constraints for an organization is that $g > 0$.*

Definition 2 *let r be the cardinality of $\boldsymbol{R}$ ($r = |\boldsymbol{R}|$), a part of the validity constraints for an organization is that $r > 0$.*

Definition 3 *let a be the cardinality of $\boldsymbol{\omega_A}$ ($a = |\boldsymbol{\omega_A}|$), a part of the validity constraints for an organization is that $a > 0$.*

Definition 4 *let p be the cardinality of $\boldsymbol{P_\Phi}$ ($p = |\boldsymbol{P_\Phi}|$). In this analysis, $p = 0$.*

Definition 5 *let $r_{avg}g$ be the average number of roles that can achieve each goal, a part of the viability constraints for an organization is that $r_{avg}g > 0$.*

We define two lemmas that are used in the analysis of the two versions of algorithm. In addition, this thesis uses the notion $\Omega(n)$ for the lower bound complexity, $\Theta(n)$ for the exact complexity, and $O(n)$ for the upper bound complexity. Furthermore, sometimes we use the terms "best case" and "worst case" for best case inputs and worst case inputs.

A power set function requires a set as the input. Given a set (s) of size n, the power set of s has a size of 2^n. The power set function will create 2^n elements. Creating a single element of the power set takes some constant time c, thus creating 2^n elements will take $\Theta(c \times 2^n)$ time. However, c is a constant factor and can be factored out. Thus, we arrive at the Lemma 1.

Lemma 1 *The time and space complexity of the power set function is* $\Theta(2^n)$.

The combinations function requires a set of sets (s) as the input. Given a set of sets (s) of size n, there are exactly $\prod_{i=1}^{n} |s_i|$ number of combinations, where $|s_i|$ is the cardinality of i^{th} element from s. Selecting one combination requires selecting one element from s_i for every element of s. So, selecting one combination takes $\Theta(n)$ time. Therefore, selecting all combinations will require $\Theta(n \times \prod_{i=1}^{n} |s_i|)$ time. Thus, we arrive at Lemma 2.

Lemma 2 *The time complexity of the combinations function is* $\Theta(n \times \prod_{i=1}^{n} |set_i|)$.

4.1 Analysis of Centralized Brute Force

Claim: Time complexity of the *CBF* for the best case is $\Theta(2^{g \times r_{avg}g})$ and for the worst case is $\Theta(2^{g \times r_{avg}g \times a})$. ■

Proof: *We show that the time complexity of the* CBF *is* $\Theta(2^{g \times r_{avg}g})$ *for the best case and* $\Theta(2^{g \times r_{avg}g \times a})$ *for the worst case.*

The pseudo code from Figure 3.1 is used for this analysis. The *CBF* is broken down into four parts for analysis.

We begin by analyzing the first part. Figure 4.1 shows the first part of the *CBF* that we are analyzing. The outer loop iterates $\Theta(g)$ time. However, the inner loop iteration is variable because the number of roles is dependent on $\boldsymbol{\omega_{G_g}}$, that is each $\boldsymbol{\omega_{G_g}}$ has a set roles that can achieve $\boldsymbol{\omega_{G_g}}$. As such, we look at two possibilities: the best case and the worst case. For the best case, each $\boldsymbol{\omega_{G_g}}$ has only one role that can achieve $\boldsymbol{\omega_{G_g}}$. For the worst case, each $\boldsymbol{\omega_{G_g}}$ can be achieved by r roles, which is all the roles available. Thus, the best case time complexity is $\Theta(g)$ and the worst case time complexity is $\Theta(g \times r)$. Similarly, the space complexity of the best case is $\Omega(g)$ and the worst case is $O(g \times r)$. By using the $r_{avg}g$, we can combine the best case and the worst case complexity. In the best case, where $r_{avg}g = 1$, $\Theta(g \times r_{avg}g) \equiv \Theta(g)$. In the worst case, where $r_{avg}g = r$, $\Theta(g \times r_{avg}g) \equiv \Theta(g \times r)$. Thus, we have $\Theta(g \times r_{avg}g)$, where $1 \leq r_{avg}g \leq r$, for both time and space complexity.

```
for each ω_Gg do
  for each R_r that achieves ω_Gg do
    maps ∪ ⟨R_r, ω_Gg⟩
  end for
end for
```

Figure 4.1: CBF Part One

Figure 4.2 shows the second part of the *CBF* that we are analyzing. From Lemma 1, generation of a power set takes $\Theta(2^n)$ time, where n is the size of the given set. Again, n in this case is dependent on the space complexity from part one. We have that the space complexity from part one is $\Theta(g \times r_{avg}g)$. So, the time taken to generate the power set is $\Theta(2^{g \times r_{avg}g})$, and the space complexity is $\Theta(2^{g \times r_{avg}g})$. Since we have assumed that there are no policies, the `reduce()` function does not do anything. Then the time complexity of the `reduce()` is $\Theta(p \times 2^{g \times r_{avg}g}) = 0$ since $p = 0$. Thus, there are no additional time complexity or reduction in space complexity.

```
λ ← reduce(powerset(maps))
```

Figure 4.2: CBF Part Two

In Figure 4.3, we see the third part of the *CBF*. The outer loop iterates for $\Theta(a)$ time. The time complexity of the inner loop depends on the space complexity from part two. We have that the space complexity from part two is $\Theta(2^{g \times r_{avg}g})$. So, the inner loop iterates for $\Theta(2^{g \times r_{avg}g})$ time. Thus, the time complexity is $\Theta(2^{g \times r_{avg}g})$. However, since there is a check in the inner loop to determine if we should create the link or not, the space complexity may be different. Again, there are two cases to look at: the best case and the worst case. For the best case, only one element from the power set passes the check (that means that every agent is capable of playing at most one role). So, we have $\Theta(a)$ for the best case space complexity. For the worst case, every element from the power set passes the check (that means that every agent is capable of playing every role required by all the goals from $\boldsymbol{\omega_G}$). So, we have $\Theta(a \times 2^{g \times r_{avg}g})$ for the worst case space complexity. Thus, we have $\Theta(a)$ space complexity for the best case and $\Theta(a \times 2^{g \times r_{avg}g})$ space complexity for the worst case.

```
for each ω_Aa do
  for each λ_m do
    if ω_Aa is capable of λ_m then
      links ∪ ⟨ω_Aa, λ_m⟩
    end if
  end for
end for
```

Figure 4.3: CBF Part Three

Finally, Figure 4.4 shows the fourth and last part of the *CBF*. Again, since we have no policies, the `reduce()` function does not do anything. Thus, there are no additional time complexity or reduction in space complexity. From Lemma 2, the `combination()` function takes $\Theta(n \times \prod_{i=1}^{n} |links_i.set|)$, where n is the number of links and $|links_i.set|$ is the cardinality of $links_i.set$. The number of links is the number of agents. $|links_i.set|$ depends on the space complexity from part four, which is $\Theta(a)$ for the best case and $\Theta(a \times 2^{g \times r_{avg} g})$ for the worst case. For the best case, the time complexity is $\Theta(a)$ because there is only one element in each $links_i.set$ (which means that there is only one combination to pick). For the worst case, the time complexity is $\Theta(a \times 2^{g \times r_{avg} g \times a})$ because there are $2^{g \times r_{avg} g}$ elements in each $links_i.set$ (which means that there are $2^{g \times r_{avg} g \times a}$ combinations to pick). Thus, the time complexity is $\Theta(a)$ for the best case and $\Theta(a \times 2^{g \times r_{avg} g \times a})$ for the worst case.

```
combinations ← combination(reduce(links))
for each combination_i from combinations do
  Φ ← {}
  for each link_l from links do
    Φ ∪ ⟨link_l.agent, link_l.set_combination_i⟩
  end for
  if P_Φ(Φ) is valid then
    if (score ← oaf(Φ)) > best.score then
      best ← ⟨score, Φ⟩
    end if
  end if
end for
```

Figure 4.4: CBF Part Four

We have the following time complexity for the *CBF*. In the best case, we have $\Theta(g \times r_{avg}g + 2^{g \times r_{avg}g} + 2^{g \times r_{avg}g} + a)$. Asymptotically, $\Theta(g \times r_{avg}g + 2^{g \times r_{avg}g} + 2^{g \times r_{avg}g} + a)$ $\equiv \Theta(2^{g \times r_{avg}g})$. The terms $g \times r_{avg}g$, one of the $2^{g \times r_{avg}g}$, and a can be dropped since $a > 0$, $g > 0$, and $r_{avg}g > 0$, which leaves the term $2^{g \times r_{avg}g}$ as the greatest contributor to the time complexity. Similarly, the worst case time complexity is $\Theta(g \times r_{avg}g + 2^{g \times r_{avg}g} + 2^{g \times r_{avg}g} + a \times 2^{g \times r_{avg}g \times a}) \equiv \Theta(a \times 2^{g \times r_{avg}g \times a})$. The terms $g \times r_{avg}g$ and $2^{g \times r_{avg}g}$ can be dropped because the term $a \times 2^{g \times r_{avg}g \times a}$ is the greatest contributor to the time complexity. Furthermore, since a also appears in the exponent, $\Theta(a \times 2^{g \times r_{avg}g \times a}) \equiv \Theta(2^{g \times r_{avg}g \times a})$.

Therefore, the time complexity of the *CBF* is $\Theta(2^{g \times r_{avg}g})$ for the best case and $\Theta(2^{g \times r_{avg}g \times a})$ for the worst case. ■

4.2 Analysis of Distributed Brute Force

There is an additional assumption required for the analysis of the distributed algorithm.

- Message delivery time takes at most $O(\epsilon)$. What that means is that from the time a message is sent, it would take at most ϵ time for the message to be received.

Claim: Time complexity of the *DBF* for the best case is $O((g \times r_{avg}g) + \lceil \frac{2^{g \times r_{avg}g}}{a} \rceil + \epsilon)$ and for the worst case is $O(2^{g \times r_{avg}g \times a} + \epsilon)$. ■

Proof: *We show that the time complexity of the* DBF *is* $O((g \times r_{avg}g) + \lceil \frac{2^{g \times r_{avg}g}}{a} \rceil + \epsilon)$ *for the best case and* $O(\epsilon + 2^{g \times r_{avg}g \times a})$ *for the worst case.*

Analysis of the *DBF* follows the pseudo code from Figure 3.4. The *DBF* analysis is broken down into five parts.

We start by analyzing the first part. Figure 4.5 shows the first part of the *DBF* that we are analyzing. The outer loop iterates $\Theta(g)$ time. However, the inner loop iteration is variable because the number of iterations is dependent on the number of roles of $\boldsymbol{\omega}_{\boldsymbol{G}g}$. As such, there are two possibilities to look at: the best case and the

worst case. In the best case, each $\boldsymbol{\omega_{Gg}}$ has only one role that can achieve $\boldsymbol{\omega_{Gg}}$. In the worst case, each $\boldsymbol{\omega_{Gg}}$ can be achieved by r roles, which is all the roles. Thus, the best case time complexity is $\Theta(g)$ and the worst case time complexity is $\Theta(g \times r)$. By using the $r_{avg}g$, we can combine the best case and worst case time complexity. In the best case, where $r_{avg}g = 1$, $\Theta(g \times r_{avg}g) \equiv \Theta(g)$. For the worst case, where $r_{avg}g = r$, $\Theta(g \times r_{avg}g) \equiv \Theta(g \times r)$. Thus, we have $\Theta(g \times r_{avg}g)$, where $1 \leq r_{avg}g \leq r$, for the time complexity.

However, the space complexity may be different because the creation of the data depends on whether $\boldsymbol{\omega_{Aa}}$ is capable of playing $\boldsymbol{R}_r$. So, we have two possibilities to look at: the best case and the worst case. If every $\boldsymbol{\omega_{Aa}}$ is not capable of playing any of the roles required by the goals in $\boldsymbol{\omega_G}$, then we have $\Theta(0)$ as the best case space complexity. What that means in this case is that agents can only do one thing which is to "do nothing". However, if every $\boldsymbol{\omega_{Aa}}$ is capable of playing r roles, then we have $\Theta(g \times r_{avg}g)$ as the worst case space complexity.

```
for each ω_Gg do
  for each R_r that achieves ω_Gg do
    if ω_Aself is capable of R_r then
      maps ∪ ⟨R_r, ω_Gg⟩
    end if
  end for
end for
```

Figure 4.5: DBF Part One

Figure 4.6 shows the second part of the *DBF* that we are analyzing. From Lemma 1, generation of a power set takes 2^n time, where n is the size of the given set. Again, n in this case is dependent on the space complexity from part one. We have that the space complexity from part one is $\Theta(0)$ for the best case and $\Theta(g \times r_{avg}g)$ for the worst case. In the best case, the time taken to generate the power set is $\Theta(2^0)$, and the worst case is $\Theta(2^{g \times r_{avg}g})$. The two time complexity can be combined to form $\Theta(\lceil \frac{2^{g \times r_{avg}g}}{a} \rceil)$. Similarly, the space complexity is $\Theta(1)$ for the best case and $\Theta(2^{g \times r_{avg}g})$ for the worst case. Since we have assumed that there are no policies, the `reduce()`

function does not do anything. Thus, there are no additional time complexity or reduction in space complexity.

$\boldsymbol{\lambda}_{self} \leftarrow$ reduce(powerset($maps$))

Figure 4.6: DBF Part Two

In Figure 4.7, we see the third part of the *DBF* that we are analyzing. Because we have assumed that there is no multicast system for sending messages, messages are sent individually to each agent. Since there are a agents, sending $\boldsymbol{\lambda}_{self}$ to all agents requires $\boldsymbol{\omega}_{\boldsymbol{A}a}$ to send $a-1$ messages. Thus, the time complexity is $\Theta(a)$. There is no additional space complexity for this part because no new data is created.

send $\boldsymbol{\lambda}_{self}$ to all agents

Figure 4.7: DBF Part Three

The fourth part of the *DBF* is shown in Figure 4.8. Because we have assumed that there is no multicast system for message sending and every agent sends a message to every other agent, that means that $\boldsymbol{\omega}_{\boldsymbol{A}a}$ receives exactly $a-1$ messages since we have a agents. Thus, the time complexity for receiving $\boldsymbol{\lambda}_i$ from all agents is $O(a+\epsilon)$. The space complexity depends on the size of $\boldsymbol{\lambda}_i$ of each agent, which we have from part two as $\Theta(1)$ for the best case and $\Theta(2^{g \times r_{avg} g})$ for the worst case. Thus, the space complexity for the best case is $\Theta(a)$ and the worst case is $\Theta(a \times 2^{g \times r_{avg} g})$.

receive $\boldsymbol{\lambda}_i$ from all agents

Figure 4.8: DBF Part Four

Finally, Figure 4.9 shows the fifth and last part of the *DBF*. From Lemma 2, the `combination()` function takes $\Theta(\prod_{i=1}^{n} |\boldsymbol{\lambda}_i|)$, where n is the number of mappings and $|\boldsymbol{\lambda}_i|$ is the cardinality of $\boldsymbol{\lambda}_i$. The number of mappings is also the number of agents. $|\boldsymbol{\lambda}_i|$ depends on the space complexity from part four, which is $\Theta(a)$ for the best case and $\Theta(a \times 2^{g \times r_{avg} g})$ for the worst case. Thus, we have the following. In the best case, the time complexity is $\Theta(a)$ because there is only one element in each $\boldsymbol{\lambda}_i$ (which

means that there is only one combination to pick). And in the worst case, the time complexity is $\Theta(a \times 2^{g \times r_{avg} g \times a})$ because there are $2^{g \times r_{avg} g}$ elements in each $\boldsymbol{\lambda}_i$ (which means that there are $2^{g \times r_{avg} g \times a}$ combinations to pick).

```
combinations ← combination(λ)
for each combination_i from combinations do
  Φ ← {}
  for each λ_j from λ do
    Φ ∪ λ_{j_combination_i}
  end for
  if P_Φ(Φ) is valid then
    if (score ← oaf(Φ)) > best.score then
      best ← ⟨score, Φ⟩
    end if
  end if
end for
```

Figure 4.9: DBF Part Five

Thus, we have the following for time complexity. In the best case, we have $O((g \times r_{avg} g)+(\lceil \frac{2^{g \times r_{avg} g}}{a} \rceil)+a+(a+\epsilon)+a)$. Asymptotically, $O((g \times r_{avg} g)+(\lceil \frac{2^{g \times r_{avg} g}}{a} \rceil)+a+(a+\epsilon)+a) \equiv O((g \times r_{avg} g)+\lceil \frac{2^{g \times r_{avg} g}}{a} \rceil+\epsilon)$. The terms a can be dropped since the a terms does not change the asymptotic complexity. Similarly, the worst case time complexity is $O((g \times r_{avg} g)+(\lceil \frac{2^{g \times r_{avg} g}}{a} \rceil)+a+(a+\epsilon)+(a \times 2^{g \times r_{avg} g \times a})) \equiv O((a \times 2^{g \times r_{avg} g \times a})+\epsilon)$. The terms $g \times r_{avg} g$, $\lceil \frac{2^{g \times r_{avg} g}}{a} \rceil$, and both a can be dropped because the term $a \times 2^{g \times r_{avg} g \times a}$ is the greatest contributor to the time complexity. Furthermore, since a also appears in the exponent, $O((a \times 2^{g \times r_{avg} g \times a}) + \epsilon) \equiv O(2^{g \times r_{avg} g \times a} + \epsilon)$.

Therefore, the time complexity of the *DBF* is $O((g \times r_{avg} g) + \lceil \frac{2^{g \times r_{avg} g}}{a} \rceil + \epsilon)$ for the best case and $O(2^{g \times r_{avg} g \times a} + \epsilon)$ for the worst case. ■

4.3 Communication Complexity

Currently, there are no communication costs associated with the *CBF* algorithm because there is no need for the algorithm to perform any sort of communication when computing the optimal organizational score. However, in the *DBF* algorithm,

there is a need for communication; the sending of an agent's mappings set ($\boldsymbol{\lambda}$) to all other agents.

Since we assumed that any message can be sent without breaking the message into multiple parts, regardless of the size of the message. The communication complexity at this stage is a linear complexity based on the number of messages sent. However, if a "real" broadcast system is available, only one message needs to be sent to achieve broadcast.

4.4 Summary

In summary, this chapter shows the time complexity of the two versions of the algorithm. The *CBF* has a time complexity of $\Theta(2^{g \times r_{avg} g})$ for the best case and $\Theta(2^{g \times r_{avg} g \times a})$ for the worst case. The *DBF* has a time complexity of $O((g \times r_{avg} g) + \lceil \frac{2^{g \times r_{avg} g}}{a} \rceil + \epsilon)$ for the best case and $O(2^{g \times r_{avg} g \times a} + \epsilon)$ for the worst case. Furthermore, with the communication assumptions, the communication complexity increases linearly in relation to the number of agents. The next chapter looks at an implementation of the two versions in a high-level simulator.

Chapter 5

Implementation

This chapter is about the implementation of the algorithm in a simulator called *Cooperative Robots Organization Simulator* (*CROS*). Furthermore, the organization aspect of *CROS* is built on the current implementation of OMACS. As of the time of this writing, most of OMACS have been implemented with the exception of policies. Thus, the implementation of the algorithm excludes assignment policies as specified by the `reduce()` function.

5.1 Cooperative Robots Organization Simulator

CROS is a high-level multi-threaded simulator. Figure 5.1 shows a simple overview of the classes used in the simulator. In *CROS*, every object is of the type AbstractObject. Physically, each object is of the same size and occupies exactly one location. A location is represented by two integers: the x-coordinate and the y-coordinate. There is no notion of a z-axis in *CROS*. However, in the future, *CROS* may introduce the notion of levels or floors. The reason for a simplistic physics environment is so that simulations of organizations-based reasoning can be tested without worrying too much about the physics of the real world.

In *CROS*, the AbstractObject represents inanimate objects within the environment. In order to allow animated objects, *CROS* provides the AbstractAgent. AbstractAgent is a subtype of AbstractObject. This is where the multi-threaded aspect of *CROS* comes into play. Every AbstractAgent that exists in the environment is a

thread that is executed once or repeatedly, depending on the code provided by the programmer. However, an AbstractAgent by itself is unable to interact with the environment. Interactions in the environment are accomplished through capabilities. *CROS* provides the AbstractCapability as the means to interact within the environment.

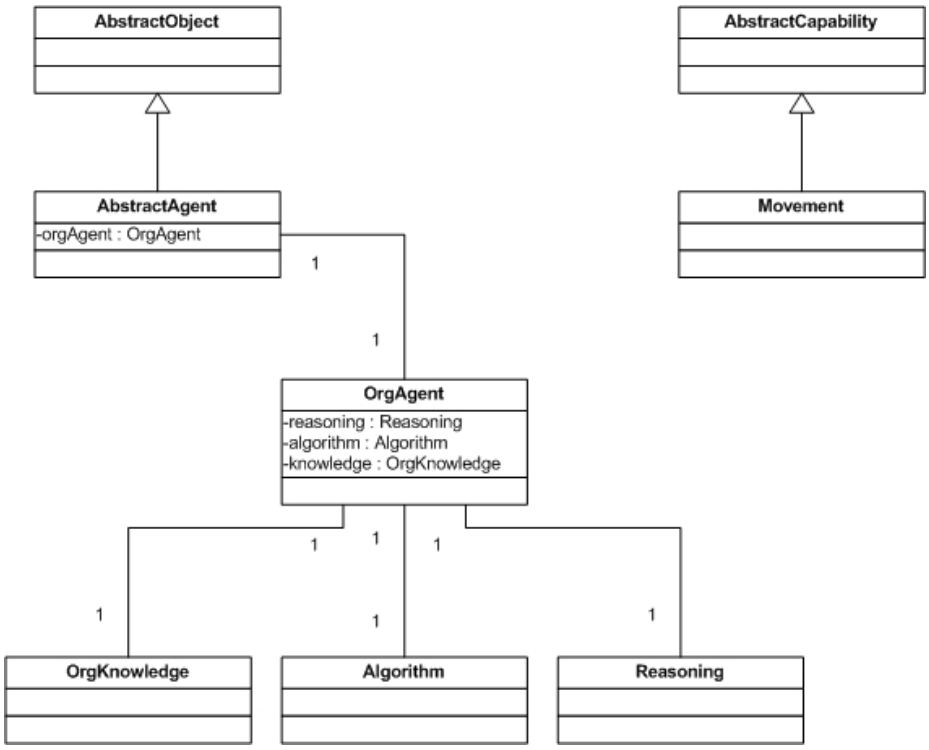


Figure 5.1: Class Diagram

The passage of time is measured in terms of turns. Each turn, certain interactions can be performed only a specified number of times. Which brings us to the two kinds of AbstractCapability: some can be used a finite number of times per turn, and some can be used an infinite number of times per turn. Transition of turns is achieved through capabilities that can be used a finite number of times per turn. These type of capabilities are known as atomic capabilities. For instance, there is a Movement capability that can only be performed once per turn. So, if a thread attempts use the Movement capability twice in a turn, the second use will automatically be put on hold until the next turn.

Furthermore, *CROS* is dependent on the current implementation of OMACS. *CROS* relies on that implementation to provide the ability for AbstractAgent to perform organization-based reasoning based on OMACS. Currently, there are three modules to the organization-based reasoning: the reasoning (Reasoning), the knowledge (OrgKnowledge), and the algorithm (Algorithm). The Reasoning is where organization-based reasoning about OMACS takes place such as deciding when and how to reorganize. The OrgKnowledge is the agent's current view of organization based on OMACS. The Algorithm is the reorganization algorithm that will produce an assignment set that provides a new viable organization.

(Note that *CROS* is in still in its early development stages. Thus the class names and diagrams in this chapter may no longer be the same with the latest version of *CROS*.)

5.1.1 Where To Get CROS

CROS is available for download from a CVS repository. The host of the repository is `macr.user.cis.ksu.edu`. The repository path is `/home/cvsroot`. *CROS* is saved under the project called `OrganizationSimulator`. Furthermore, since *CROS* relies on the current implementation of OMACS, another project is required as well. This project is called `OrganizationModel` from the same CVS repository.

5.2 Algorithm Implementation

The two versions of the algorithm are implemented as the algorithm module in the organization-based reasoning module. Both versions implement the interface Algorithm. The centralized version is called `CentralizedBruteForce.java` and the distributed version is called `DistributedBruteForce.java`. They are available as part of the *CROS* project. The implementation is not exactly the same as the pseudo code listed in Chapter 3. A difference is due to the fact that policies are not implemented in the version of OMACS used in for this thesis.

5.2.1 Centralized Brute Force Implementation

The *CBF* implementation is broken into three parts. The following code has been edited to remove code used for profiling and debugging purposes.

Listing 5.1 shows the main `reorganize()` method. This method is called whenever a reorganization is required by the reasoning module.

Listing 5.1: CBF Implementation Part One

```
1 public Collection<Assignment> reorganize(Organization organization, Collection<
      GoalLeaf> workingGoals, Collection<Agent> workingAgents) {
2   List<Link> links = new ArrayList<Link>();
3   buildLinks(links, organization, workingGoals, workingAgents);
4   return findOptimalOrganization(links, organization);
5 }
```

The `buildLinks()` method is shown in Listing 5.2. This method creates the data structures required by `findOptimalOrganization()`.

Listing 5.2: CBF Implementation Part Two

```
6 private void buildLinks(List<Link> links, Organization organization, Collection<
      GoalLeaf> goals, Collection<Agent> agents) {
7   /* create the <role, goal> mappings */
8   Set<Map> mapSet = new HashSet<Map>();
9   for (GoalLeaf<?> g : goals) {
10    for (Role r : g.getAchievedBySet()) {
11      mapSet.add(new Map(r, g));
12    }
13  }
14  /* create the powerset of the <role, goal> mappings */
15  Set<Set<Map>> powerSet = new PowerSet<Map>().powerSetT(mapSet);
16  /* TODO remove invalid sets of <role, goal> mappings */
17  /* map agents to sets of <role, goal> mappings */
18  for (Agent<?, ?> a : agents) { /* for each agent */
19    Link link = new Link(a);
20    for (Set<Map> set : powerSet) { /* for each set of <roles,goals> */
21      boolean capable = true;
22      /* check if the agent is capable of playing all the mappings */
23      for (Map map : set) {
24        capable &= (map.role.rcf(a) > 0.0);
25      }
26      if (capable) {
27        link.assignments.add(set);
28      }
```

```
29      }
30      links.add(link);
31    }
32    /* TODO remove invalid <agent, role, goal> assignments */
33  }
```

The `findOptimalOrganization()` method is shown in Listing 5.3. This method computes the optimal organizational score and returns an assignment set that gives the optimal organizational score.

Listing 5.3: CBF Implementation Part Three

```
34  private Collection<Assignment> findOptimalOrganization(List<Link> links,
        Organization organization) {
35    long numberOfCombinations = 1;
36    for (Link l : links) {
37      numberOfCombinations *= l.assignments.size();
38    }
39    double bestAssignmentScore = 0.0;
40    Collection<Assignment> bestAssignmentSet = new ArrayList<Assignment>();
41    for (long combination = 0; combination < numberOfCombinations; combination++) {
42      /* create an assignment set */
43      Collection<Assignment> assignmentSet = new ArrayList<Assignment>();
44      for (Link l : links) {
45        for (Map m : l.assignments.get(l.index)) {
46          assignmentSet.add(new Assignment(l.agent, m.role, m.goal));
47        }
48      }
49      /* determine oaf() score */
50      organization.clearAssignmentSet();
51      organization.addAssignmentSet(assignmentSet);
52      double score = organization.oaf();
53      if (score > bestAssignmentScore) {
54        /* better organization */
55        bestAssignmentScore = score;
56        bestAssignmentSet = assignmentSet;
57      }
58      /* go to next combination */
59      for (int i = 0; i < links.size(); i++) {
60        /* increment the index of the ith element */
61        Link l = links.get(i);
62        l.index++;
63        if (l.index < l.assignments.size()) {
64          break;
65        } else {
```

```
66            l.index = 0;
67          }
68        }
69      }
70      /* at this point, we have the best organization, if one exists */
71      organization.clearAssignmentSet();
72      organization.addAssignmentSet(bestAssignmentSet);
73      return bestAssignmentSet;
74  }
```

5.2.2 Distributed Brute Force Implementation

The *DBF* is broken into five parts. The following code has been edited to remove code used for profiling and debugging purposes.

Listing 5.4 shows the main `reorganize()` method. This method is called whenever a reorganization is required by the reasoning module.

Listing 5.4: DBF Implementation Part One

```
1  public Collection<Assignment> reorganize(Organization organization, Collection<
       GoalLeaf> working, Collection<Agent> workingAgents) {
2    List<Set<Map>> maps = new ArrayList<Set<Map>>();
3    buildMaps(maps, organization, working);
4    send(maps, organization);
5    List<Link> links = new ArrayList<Link>();
6    links.add(new Link(organization.getAgent(parent.name), maps));
7    receive(links, organization);
8    return findOptimalOrganization(links, organization);
9  }
```

Listing 5.5 shows the `buildMaps()` method. This method creates the data structures required by the `send()` method.

Listing 5.5: DBF Implementation Part Two

```
10 private void buildMaps(List<Set<Map>> maps, Organization organization, Collection
       <GoalLeaf> goals) {
11   /* create the <role, goal> mappings */
12   Set<Map> mapSet = new HashSet<Map>();
13   for (GoalLeaf<?> g : goals) {
14     for (Role r : g.getAchievedBySet()) {
15       mapSet.add(new Map(r, g));
16     }
```

```
17    }
18    /* create the powerset of the <role, goal> mappings */
19    Set<Set<Map>> powerSet = new PowerSet<Map>().powerSetT(mapSet);
20    /* TODO remove invalid sets of <role, goal> assignments */
21    /* map agents to sets of <role, goal> assignments */
22    for (Set<Map> set : powerSet) { /* for each set of <roles,goals> */
23      boolean capable = true;
24      /* check if the agent is capable of playing all the assignments */
25      for (Map map : set) {
26        capable &= (map.role.rcf(organization.getAgent(parent.name)) > 0.0);
27      }
28      if (capable) {
29        maps.add(set);
30      }
31    }
32  }
```

Listing 5.6 shows the `send()` method. This method sends the agent's data over to the other agents.

Listing 5.6: DBF Implementation Part Three

```
33  private void send(List<Set<Map>> maps, Organization organization) {
34    for (Agent<?, String> a : organization.getAgentSet()) {
35      parent.send(new Message(parent.name, a.getContactInformation(), "",
36          maps));
37    }
38  }
```

Listing 5.7 shows the `receive()` method. This method constructs the necessary data structure required by the `findOptimalOrganization()` by receiving data from every other agent.

Listing 5.7: DBF Implementation Part Four

```
39  private void receive(List<Link> links, Organization organization) {
40    while (parent.messages() != organization.getAgentSet().size()) {
41      parent.endTurn();
42    }
43    for (Message m = (Message) parent.receive(); m != null; m = (Message) parent
44        .receive()) {
45      links.add(new Link(organization.getAgent(m.sender),
46          (List<Set<Map>>) m.content));
47    }
48  }
```

Listing 5.8 shows the `findOptimalOrganization()` method. This method computes the optimal organizational score and returns an assignment set that gives the optimal organizational score.

Listing 5.8: DBF Implementation Part Five

```
49 private Collection<Assignment> findOptimalOrganization(List<Link> links,
       Organization organization) {
50   long numberOfCombinations = 1;
51   for (Link l : links) {
52     numberOfCombinations *= l.assignments.size();
53   }
54   double bestAssignmentScore = 0.0;
55   Collection<Assignment> bestAssignmentSet = new ArrayList<Assignment>();
56   for (long combination = 0; combination < numberOfCombinations; combination++) {
57     /* create an assignment set */
58     Collection<Assignment> assignmentSet = new ArrayList<Assignment>();
59     for (Link l : links) {
60       for (Map m : l.assignments.get(l.index)) {
61         assignmentSet.add(new Assignment(l.agent, m.role, m.goal));
62       }
63     }
64     /* determine oaf() score */
65     organization.clearAssignmentSet();
66     organization.addAssignmentSet(assignmentSet);
67     double score = organization.oaf();
68     if (score > bestAssignmentScore) {
69       /* better organization */
70       bestAssignmentScore = score;
71       bestAssignmentSet = assignmentSet;
72     }
73     /* go to next combination */
74     for (int i = 0; i < links.size(); i++) {
75       /* increment the index of the ith element */
76       Link l = links.get(i);
77       l.index++;
78       if (l.index < l.assignments.size()) {
79         break;
80       } else {
81         l.index = 0;
82       }
83     }
84   }
85   /* at this point, we have the best organization, if one exists */
86   organization.clearAssignmentSet();
```

```
87    organization.addAssignmentSet(bestAssignmentSet);
88    return bestAssignmentSet;
89  }
```

5.3 Summary

In summary, this chapter highlights the implementation of the two versions of the algorithm in *CROS*. The next chapter provides a look at some compiled results of testing the implementation of the *CBF*.

Chapter 6

Results

This chapter shows the results of the *CBF*. There are three sections to this chapter: § 6.1 discusses the runtime results of the *CBF*; § 6.2 discusses the optimal assignments set produced by the *CBF*; and § 6.3 discusses some of the characteristics discovered when developing and testing the *CBF* of OMACS.

6.1 Runtime Results

To test the *CBF*, a random model generator is created for OMACS. The random model generator is available as the project `OrganizationModelGenerator` from the same CVS repository. The random model generator takes five inputs: the number of goals, the number of roles, the number of agents, the average number of roles per goal, and the average number of agents per role. From the five inputs given, a model is randomly generated.

In the test cases, the inputs were set to range from $1 \ldots 3$ for all five inputs. However, the last two inputs (average number of roles per goal, and average number of agents per role) were bounded by the number of roles and the number of agents respectively. In addition, there were ten runs of the same tests so that a better estimate of the runtime could be obtained. In the tests, the time was logged just before the algorithm begins and immediately after the algorithm completes. The time was tracked in nanoseconds.

This test suite was written in Java. The system used for testing was a Pentium

III 1.7 GHz, 1 GB RAM running Slackware 10.2. However, the precision of the JVM only provided us with microseconds.

In the first batch of tests results, the first case (one goal, one role, one agent, one average role per goal, one average agent per role) would have extremely high running time. Subsequent cases would have incremental drops in running time. The JVM was determined to be the cause of the problem after a careful analysis of the results. In the JVM, the first time a byte code is executed, that code is interpreted. And using some heuristics, that code might be compiled to native code for later executions, which will run significantly faster. Thus, in all ten samples, the original runtime of the first case was extremely high while subsequent cases had much lower running time. Furthermore, the disparity in runtime is noticeable because the early cases had a running time of hundreds of microseconds while subsequent cases had an exponentially increasing running time.

With this knowledge, modifications were made to test suite to include a "warm up" phase where the required classes are called one thousand times before the actual timing of the algorithm starts. This number forces almost all of the necessary byte code to be compiled natively, although there is no guarantee as the heuristics used for determining when byte code is compiled to native code is unknown. With the revised test suite, a second batch of ten samples were obtained and the results were more consistent with the expected results based on the analysis.

Furthermore, since the timings are in microseconds, the small cases like the first case are highly susceptible to external factors because the running time of the small cases are in hundreds of microseconds. A simple keyboard or mouse interrupt contributes significantly to the running time of the small cases. The following five graphs shows a sample of the compiled runtime results. The graphs are represented in a logarithmic scale. Figure 6.1 shows the runtime results as the number of goals increases. Figure 6.2 shows the runtime results as the number of roles increases. Figure 6.3 shows the runtime results as the number of agents increases. Figure 6.4 shows the runtime results as the average number of roles per goal increases. Figure 6.5 shows

the runtime results as the average number of agents per role increases. All figures show three samples taken from the compiled data. Each line represents a sample where the other inputs are fixed at the values of 1, 2, or 3 as shown in the figures' legends.

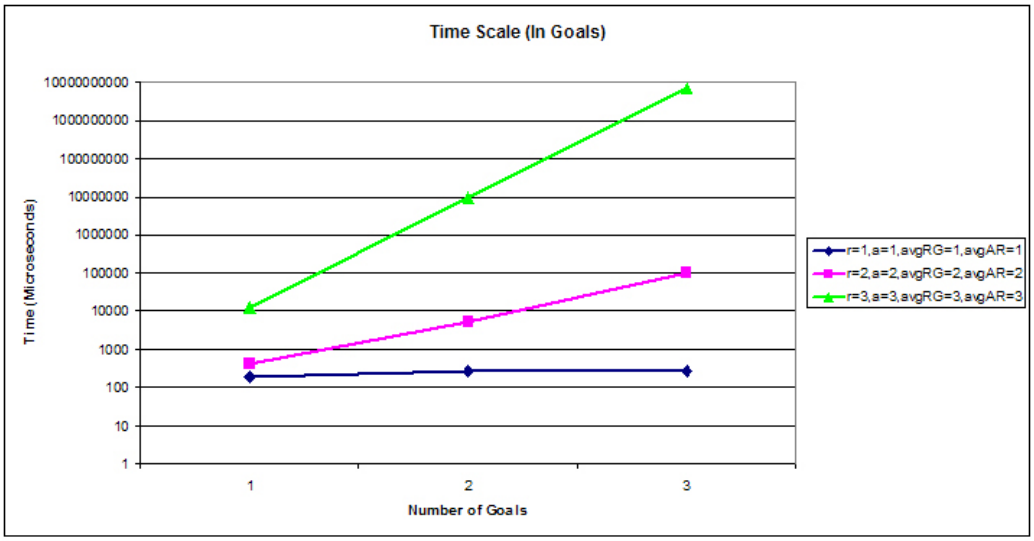


Figure 6.1: Time - Goals

The five graphs concur with the time complexity analysis of the algorithm, especially in the cases where the values of four inputs are set to 3. In the logarithmic graphs, a straight line with 0 gradient means a polynomial time complexity and a straight line with a positive gradient means an exponential time complexity. The results of the small cases (four inputs set to 1) show a slight exponential time complexity while the large cases (four inputs set to 3) show an exponential time complexity consistent with the analysis of $\Theta(2^{g \times r_{avg} g \times a})$.

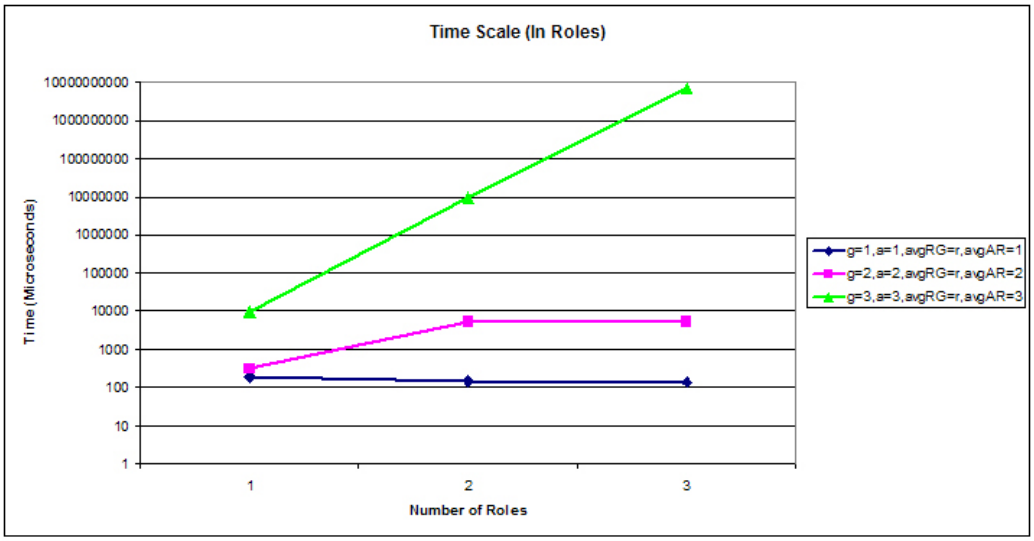


Figure 6.2: Time - Roles

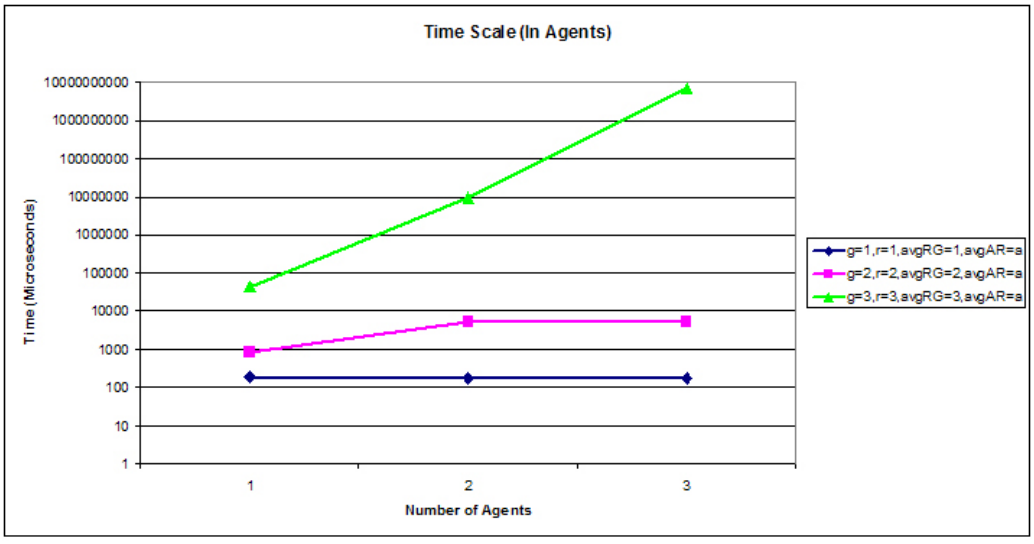


Figure 6.3: Time - Agents

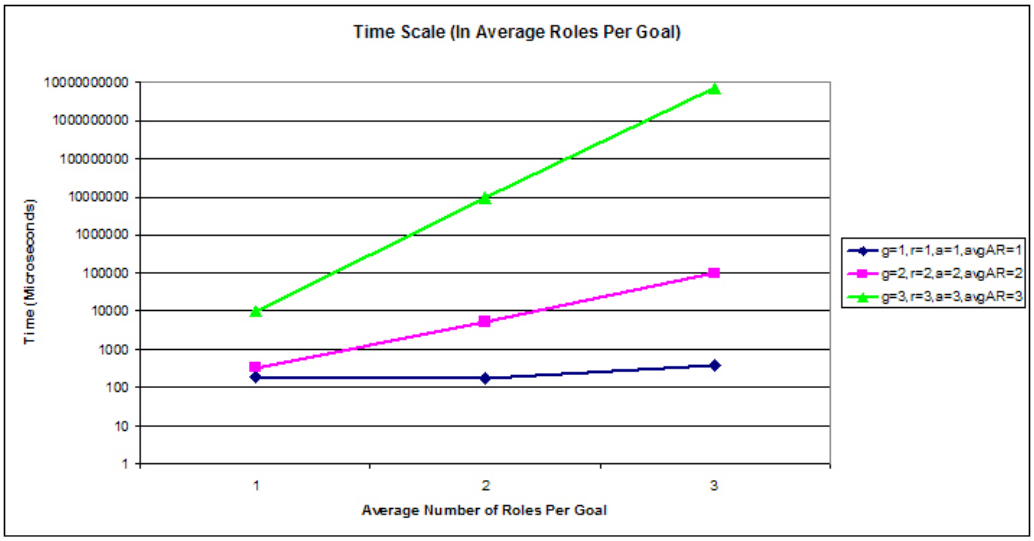


Figure 6.4: Time - Average Roles Per Goal

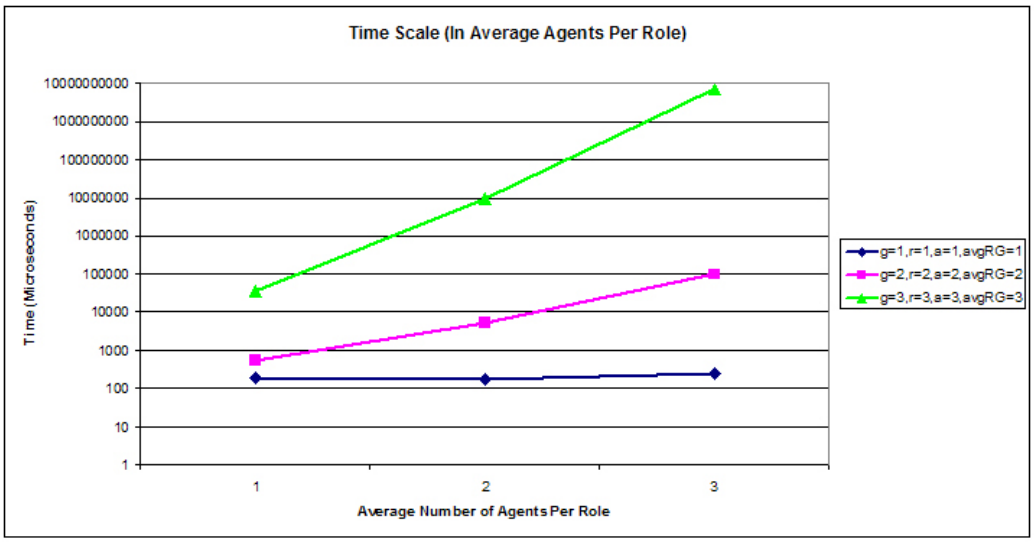


Figure 6.5: Time - Average Agents Per Role

6.2 Optimal Assignments Set

Next, the *CBF* was tested on three actual models for correct results. The three models are taken from two applications: "Adaptive Information System" (one model) and "Search and Rescue" (two models)[1]. For testing purposes, the implementations differs slightly from the models. Since the models do not show any scores for relations, all scores are assumed to have a value of 1.0 for simplicity. That means that roles achieve goals with a score of 1.0 and agents possesses capabilities with a score of 1.0.

6.2.1 Adaptive Information System Model

Figure 6.6 shows the goal model for the "Adaptive Information System". There are six leaf goals.

G1 1: Validate Enemy Vehicles

G2 2.1.1: Access Moving Vehicle Sensor

G3 2.1.2: Combine Moving Vehicles Into List

G4 2.2: Combine Moving & ID's Vehicle

G5 2.3.1: Access ID Vehicle Sensor

G6 2.3.2: Combine ID Vehicle Into List

Figure 6.7 shows the role model. Six roles are defined in the role model and for each role the capabilities required by that role. Each role is defined for each of the leaf goals. However, the figure does not indicate which goal is achieved by which role. Fortunately, there is a text description that states that each role is designed to achieve a specific goal.

R1 Enemy Vehicle Validator $\rightarrow \{G1\}$

R2 Moving Vehicle Sensor Interface $\rightarrow \{G2\}$

R3 Moving Vehicle List Combiner $\rightarrow \{G3\}$

[1]The models were taken from Dr. Scott A. DeLoach's working notes.

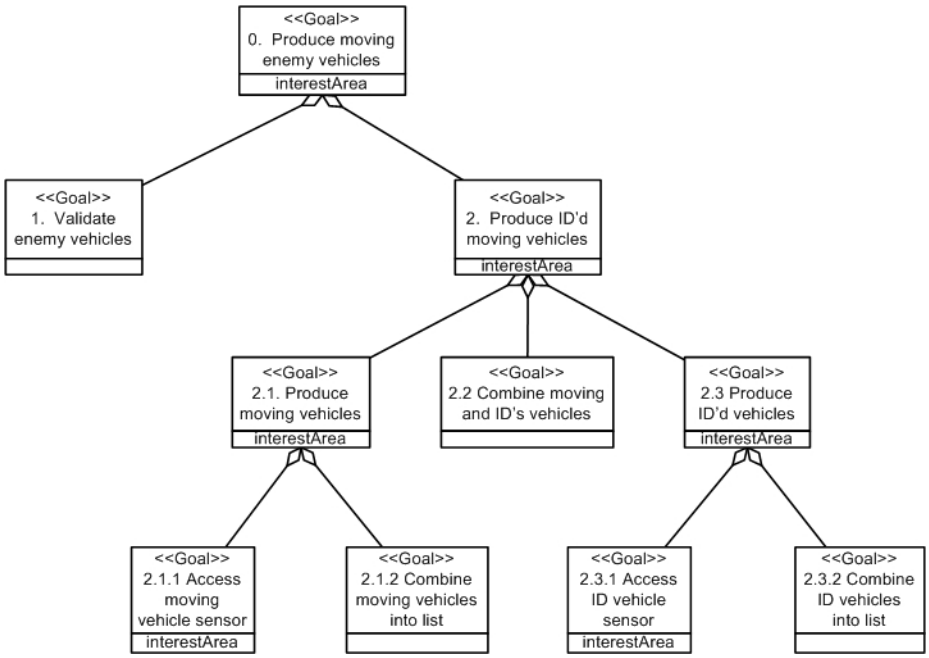


Figure 6.6: Adaptive Information Systems Goal Model

R4 Moving/ID Combiner $\rightarrow \{G4\}$

R5 ID Sensor Interface $\rightarrow \{G5\}$

R6 ID List Combiner $\rightarrow \{G6\}$

The following are the eight capabilities defined in the role model.

C1 Capable Of Validation

C2 Access To Enemy/Friendly Database

C3 Access To Moving Vehicle Sensor

C4 Area Of Coverage

C5 Capable Of Producing Moving Vehicle List

C6 Capable Of Combining Moving & ID List

C7 Access To ID Sensor

C8 Capable Of Producing Vehicle ID List

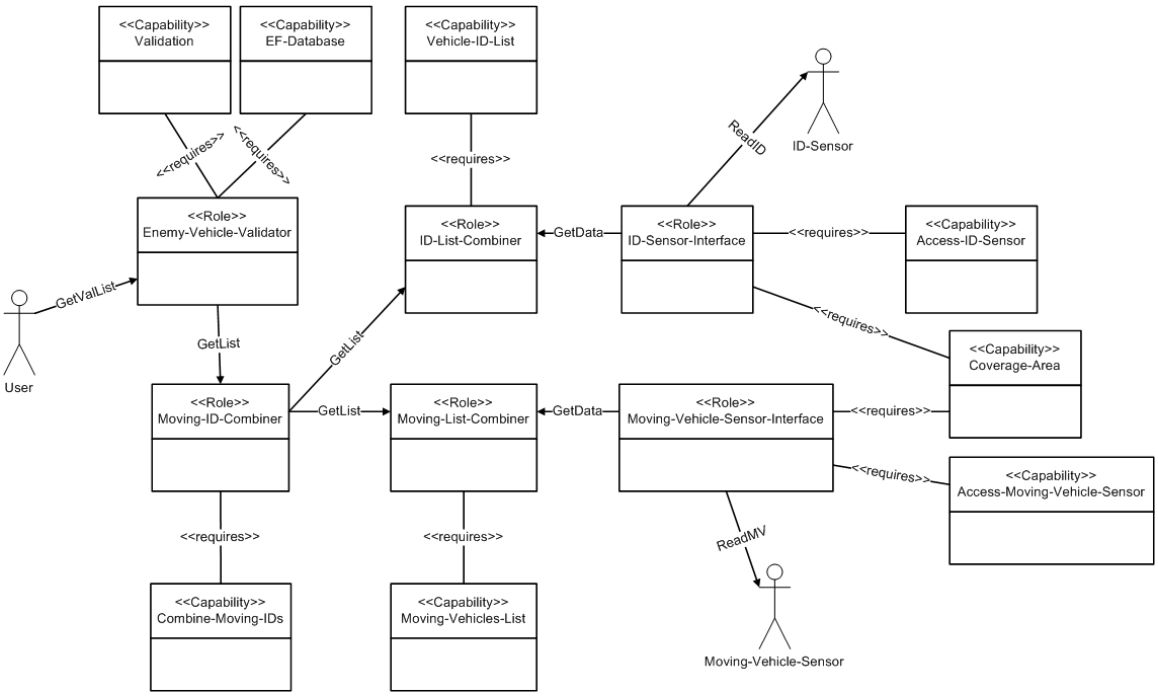


Figure 6.7: Adaptive Information Systems Role Model

Eight agents were defined as the number of agents was not specified by the models. Six agents possessing only the capabilities required by one of the roles, one agent possessing all the capabilities, and one agent possessing none of the capabilities.

A1 $\leftrightarrow \{C1, C2, C3, C4, C5, C6, C7, C8\}$

A2 $\leftrightarrow \{C1, C2\}$

A3 $\leftrightarrow \{C3, C4\}$

A4 $\leftrightarrow \{C5\}$

A5 $\leftrightarrow \{C6\}$

A6 $\leftrightarrow \{C4, C7\}$

A7 $\leftrightarrow \{C8\}$

A8 $\leftrightarrow \{\}$

Table 6.1 shows the results returned by the algorithm. Given the default `oaf()`, the optimal assignments set would be that agent $A1$ will play the roles ($R1$, $R2$, $R3$,

$R4$, $R5$, and $R6$) to achieve the goals ($G1$, $G2$, $G3$, $G4$, $G5$, and $G6$) respectively. $A2$ will play $R1$ to achieve $G1$; $A3$ plays $R2$ to achieve $G2$; $A4$ plays $R3$ to achieve $G3$; $A5$ plays $R4$ to achieve $G4$; $A6$ plays $R5$ to achieve $G5$; $A7$ plays $R6$ to achieve $G6$; and $A8$ cannot play any roles. $A1$ can play all roles because $A1$ possesses the capabilities required by every role. $A2$ can only play $R1$ because $A2$ only possesses the capabilities required for $R1$. Since no role's required capabilities set is a subset of another role's required capabilities set; $A2$, $A3$, $A4$, $A5$, $A6$, and $A7$ can only play the role they are designed for, which is $R1$, $R2$, $R3$, $R4$, $R5$, and $R6$ respectively. Comparing the expected results to the results from Table 6.1, it is clear that the implementation of the *CBF* returns the optimal result.

Agent	Role	Goal
$A1$	$R1$	$G1$
$A1$	$R2$	$G2$
$A1$	$R3$	$G3$
$A1$	$R4$	$G4$
$A1$	$R5$	$G5$
$A1$	$R6$	$G6$
$A2$	$R1$	$G1$
$A3$	$R2$	$G2$
$A4$	$R3$	$G3$
$A5$	$R4$	$G4$
$A6$	$R5$	$G5$
$A7$	$R6$	$G6$

Table 6.1: Adaptive Information Systems Assignments Set

6.2.2 Search and Rescue

The two versions of the "Search and Rescue" share the same goal model. The difference lies in their role model. In Figure 6.8, five leaf goals are defined.

G1 1.1: Search Area

G2 1.2: Identify Victims

G3 2.1: Locate Victim

G4 2.2: Pickup Victim

G5 2.3: Carry Victim Home

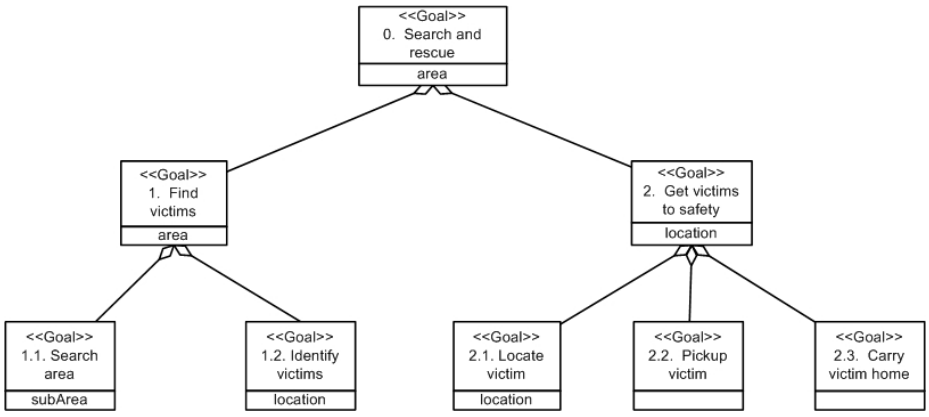


Figure 6.8: Search and Rescue Goal Model

6.2.2.1 Search and Rescue Version 1

Figure 6.9 shows the first version of the role model. Two roles are defined to achieve the five leaf goals from Figure 6.8.

R1 Searcher $\rightarrow \{G1, G2\}$

R2 Rescuer $\rightarrow \{G3, G4, G5\}$

In addition, Figure 6.9 defines four capabilities that are required the roles.

C1 Range Sensor

C2 ID Sensor

C3 Movement

C4 Gripper

Four agents were defined: two agents possessing only the capabilities required by one of the roles, one agent possessing all the capabilities, and one agent possessing none of the capabilities.

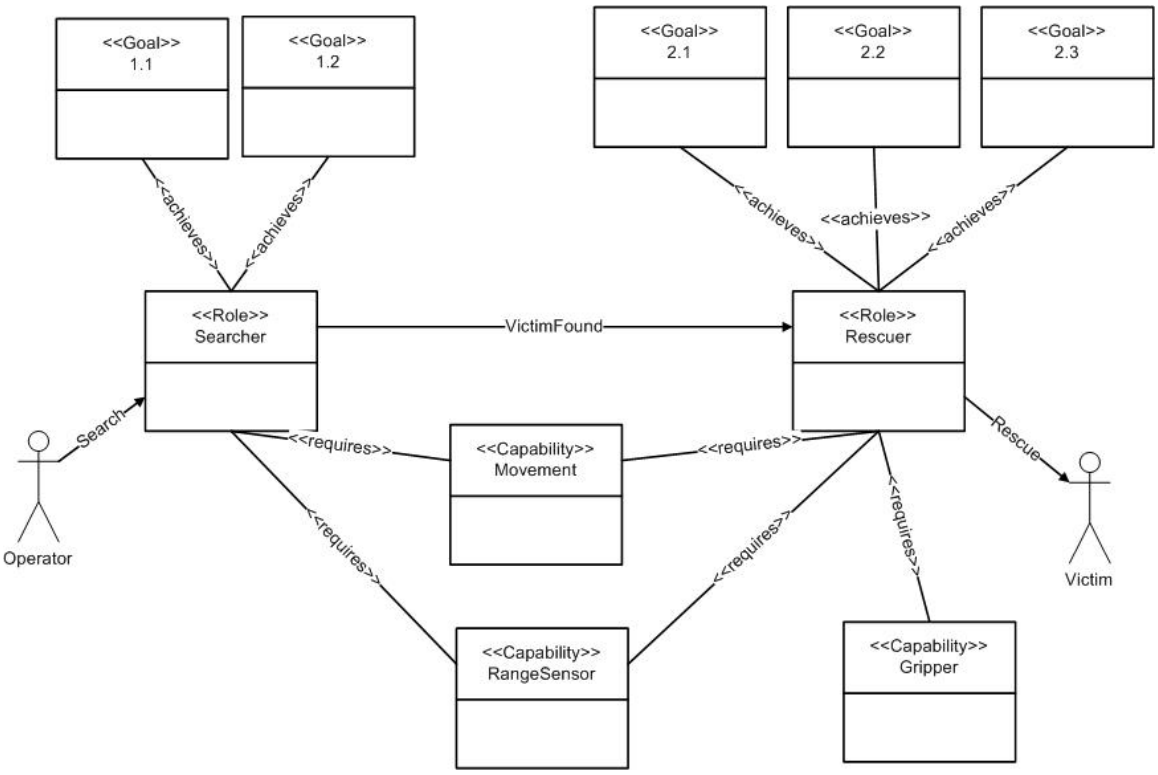


Figure 6.9: Search and Rescue Role Model Version 1

A1 $\leftrightarrow \{C1, C2, C3, C4\}$

A2 $\leftrightarrow \{C1, C2, C3\}$

A3 $\leftrightarrow \{C1, C3, C4\}$

A4 $\leftrightarrow \{\}$

Table 6.2 shows the results returned by the algorithm. Given the default `oaf()`, the optimal assignments set would be that agent $A1$ will play the roles $R1$ and $R2$ to achieve the goals $\{G1, G2\}$, and $\{G3, G4, G5\}$ respectively. $A2$ will play $R1$ to achieve $G1$ and $G2$; $A3$ plays $R2$ to achieve $G3$, $G4$, and $G5$; and $A4$ cannot play any roles. $A1$ can play all roles because $A1$ possesses the capabilities required by every role. $A2$ can only play $R1$ because $A2$ only possesses the capabilities required for $R1$. Since no role's required capabilities set is a subset of another role's required capabilities set; $A2$ and $A3$ can only play the role they are designed for, which is $R1$

and $R2$ respectively. Comparing the expected results to the results from Table 6.2, it is clear that the implementation of the *CBF* returns the optimal result.

Agent	Role	Goal
$A1$	$R1$	$G1$
$A1$	$R1$	$G2$
$A1$	$R2$	$G3$
$A1$	$R2$	$G4$
$A1$	$R2$	$G5$
$A2$	$R1$	$G1$
$A2$	$R1$	$G2$
$A3$	$R2$	$G3$
$A3$	$R2$	$G4$
$A3$	$R2$	$G5$

Table 6.2: Search and Rescue Version 1 Assignments Set

6.2.2.2 Search and Rescue Version 2

Figure 6.10 shows the second version of the role model. In addition to the two roles from the first version, an additional role is defined. More accurately, the "Searcher" role is broken down into two roles: "Searcher" and "Identifier". The types of capabilities remains the same as the first version. The only change is that the "Searcher" role no longer requires "ID Sensor" capability, that capability is now required by the "Identifier" role.

R1 Searcher $\rightarrow \{G1\}$

R2 Rescuer $\rightarrow \{G3, G4, G5\}$

R3 Identifier $\rightarrow \{G2\}$

Five agents were defined: three agents possessing only the capabilities required for one of the roles, one agent possessing all the capabilities, and one agent possessing none of the capabilities.

A1 $\leftrightarrow \{C1, C2, C3, C4\}$

A2 $\leftrightarrow \{C1, C3\}$

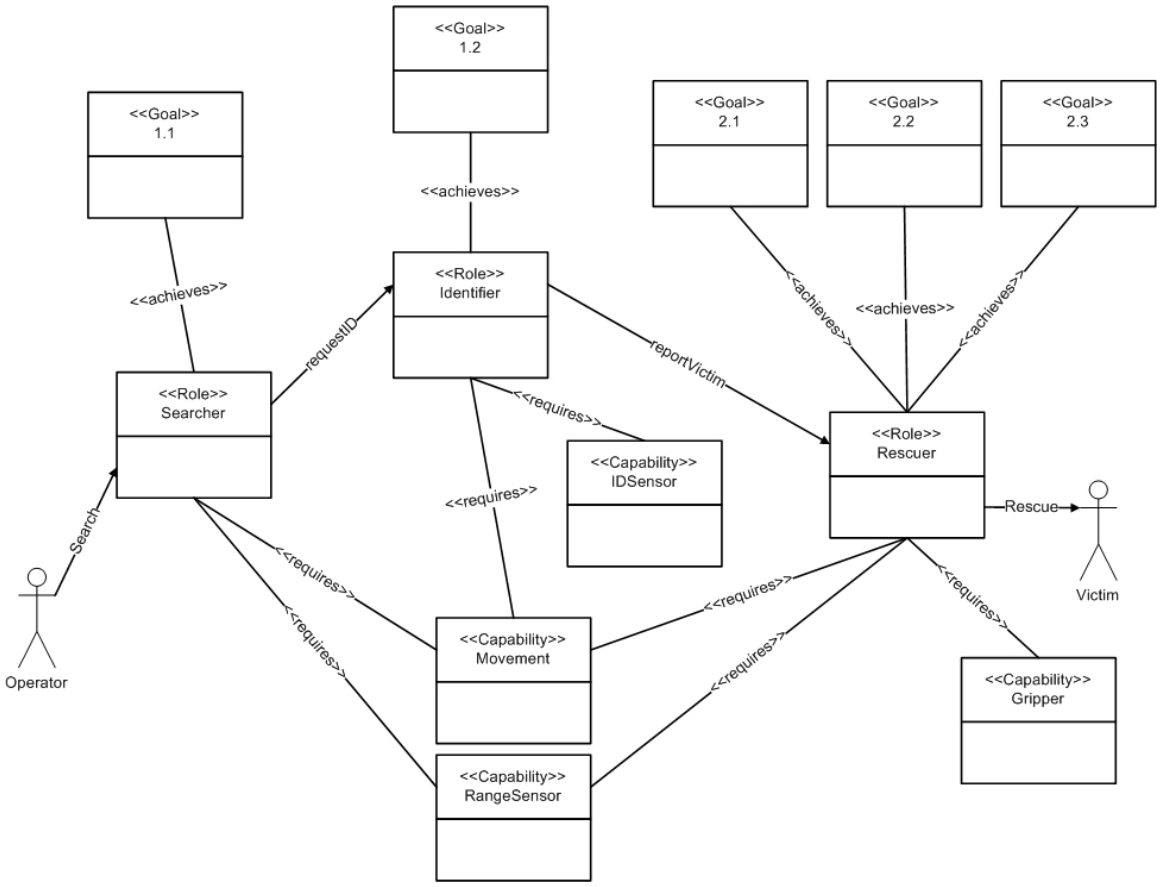


Figure 6.10: Search and Rescue Role Model Version 2

A3 $\leftrightarrow \{C1, C3, C4\}$

A4 $\leftrightarrow \{C2, C3\}$

A5 $\leftrightarrow \{\}$

Table 6.3 shows the results returned by the algorithm. Given the default `oaf()`, the optimal assignments set would be that agent $A1$ will play the roles $R1$, $R2$, and $R3$ to achieve the goals $G1$, $\{G3, G4, G5\}$, and $G2$ respectively. $A2$ will play $R1$ to achieve $G1$; $A3$ plays $R1$ and $R2$ to achieve $G1$, and $\{G3, G4, G5\}$ respectively; $A4$ plays $R3$ to achieve $G2$; and $A5$ cannot play any roles. $A1$ can play all roles because $A1$ possesses the capabilities required by every role. $A2$ can only play $R1$ because $A2$ only possesses the capabilities required for $R1$. Similarly, $A4$ can only play $R3$ because $A4$ only possesses the capabilities required for $R3$. However, it is interesting

to note that $A3$ is able to play two roles ($R1$ and $R2$) even though $A3$ was designed for $R2$. This happens because the required capabilities set of $R1$ is a subset of the required capabilities set of $R2$. Comparing the expected results to the results from Table 6.3, it is clear that the implementation of the *CBF* returns the optimal result.

Agent	Role	Goal
$A1$	$R1$	$G1$
$A1$	$R2$	$G3$
$A1$	$R2$	$G4$
$A1$	$R2$	$G5$
$A1$	$R3$	$G2$
$A2$	$R1$	$G1$
$A3$	$R1$	$G1$
$A3$	$R2$	$G3$
$A3$	$R2$	$G4$
$A3$	$R2$	$G5$
$A4$	$R3$	$G2$

Table 6.3: Search and Rescue Version 2 Assignments Set

6.3 Reducing Time Complexity

Using the time complexity derived from Chapter 4 for the *CBF*, design characteristics were uncovered that would lead to a run time that is close to the best case inputs. The best case time complexity is $\Theta(2^{g \times r_{avg}g})$ and the worst case time complexity is $\Theta(2^{g \times r_{avg}g \times a})$. Also, the effects of assignment policies on the time complexity is further discussed in § 6.3.2.

6.3.1 General Designs

Typically, when designing an organization, designers often associate a leaf goal with a role. This approach leads to a more restrictive model in terms of adaptability to a changing environment. [RDK06] provides more information on the issue of flexibility. For example, in the case of having one role per goal, the variable $r_{avg}g$ can be factored out from the complexity because $r_{avg}g = 1$. In addition, having one role per agent will

effectively reduce the number of possible assignments to exactly two per agent: one playing the role, and the other not playing any role. This would lead to a significantly reduced time complexity of $\Theta(2^a)$. However, as was mentioned earlier, this approach leads to a loss of flexibility in the model. In order to allow more flexibility without disregarding this approach, a more complicated model is required which results in an increase to the time complexity.

There are many more design characteristics that can be discovered. One such area is to look at the issue of the flexibility of a model versus the efficiency of a model. Perhaps compromise can be found to provide an acceptable efficiency and flexibility. However, this area is left open to future work.

6.3.2 Assignment Policies

Assignment policies can have varying effects on the search space of the algorithm. For instance, if there is a policy that says that "agents can only play one role at a time", the search space for each agent is reduced by a significant amount which directly affects the time complexity. Assume an agent is able to play five roles and each role achieves three goals. In the default situation, this agent has $2^{5\times3=15} = 32,768$ possible assignments. However, with this policy, the agent only has $5\times3 = 15$ possible assignments. If there are four other similar agents, without the policy there is a total of $32,768^4 = 1,152,921,504,606,846,976$ combinations. With the policy, the total combinations are $15^4 = 50,625$.

In the preliminary analysis, the CBF was modified into two versions that use the policy "agents can only play one role at a time". In the CBF, there are two locations that policy checking occurs. However, this particular policy is only applicable in the first location as noted in Figure 6.11.

$\boldsymbol{\lambda} \leftarrow$ reduce(powerset($maps$))

Figure 6.11: First Policy Check

The two versions highlights two ways the policy can be applied to the CBF algo-

rithm.

Version 1 prunes the power set by removing invalid sets. For this policy, sets with more than one element is removed.

Version 2 replaces the power set function with a custom function that only generates sets with one element.

Figure 6.12 shows the preliminary results that compare the CBF algorithm versus the two modified versions. As expected, version 1 performs worse in small cases because the power set function is the greatest contributor to the time complexity and the extra time required to remove invalid sets. However, when the power set function stops being the greatest contributor to the time complexity, version 1 begins to perform better. Version 2 on the otherhand, always performs better than both the original CBF algorithm and version 1.

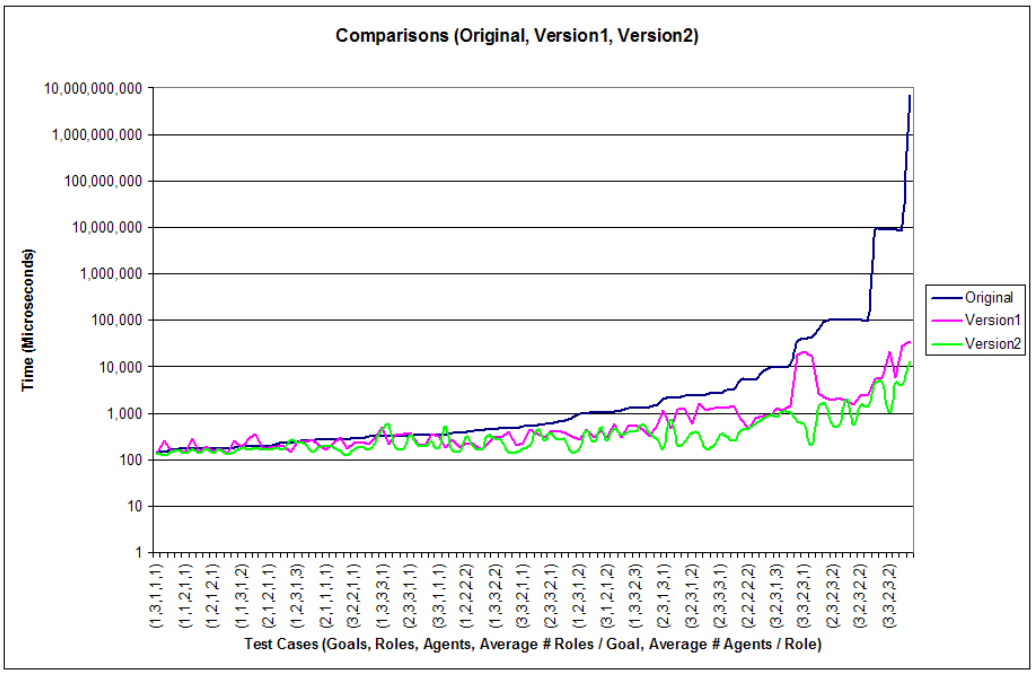


Figure 6.12: Comparisons

Similarly, a policy like "a role can only achieved one goal" will have similar effect on the time complexity. Using the above example with this policy, the possible

assignments for each agent will be $2^{5\times1=5} = 32$ and the total combinations of the four agents will be $32^4 = 1,048,576$.

Using both policies will yield a greater decrease in the time complexity. The possible assignments for an agent will be 5 and the total combinations will be $5^4 = 625$. The effects of policies on the time complexity varies based on the policies involved. In this particular case, using both policies the time complexity of the *CBF* is changed such that the greatest contributor to the time complexity is the generation of the power set instead of the combinations. However, this does not take into consideration the additional time complexity costs from performing policy checks. In Chapter 4, the analysis assumes that there are no policies so there are no additional time complexity.

Adding additional policies also adds additional time complexity. In this particular case, both policies adds a linear time complexity based on the size of the inputs. There are two reductions performed in the *CBF*: once for the power set and once for the *links*. Each policy will have to be checked against every element from the power set and the *links*. However, it is unknown as to whether the benefits of policies outweigh the additional time complexity for policy checking and also if any additional policy only adds a linear time complexity. The study of this area of policies is left open to future work.

6.4 Summary

In summary, this chapter shows the results of the implementation and some characteristics pertaining to the algorithm. The results confirm the exponential time complexity involved in finding an optimal organization score. The characteristics highlight how to reduce the time complexity of the *CBF* but will result in a less flexible model. The next chapter concludes the thesis.

Chapter 7

Conclusions

This chapter concludes the thesis by summarizing the work accomplished. In addition, potential areas for future work are highlighted.

7.1 Thesis Contributions

The work done in this thesis lays the groundwork for future research into more efficient reorganization algorithms for OMACS. This thesis introduces an reorganization algorithm that produces an optimal solution and analyzes the complexity of the reorganization algorithm, which shows an exponential time complexity in both the centralized and distributed version. The exponential time complexity is a result of limitations in OMACS that does not allow more information to be obtained with respect to reorganization algorithms. The exponential time complexity prevents the reorganization algorithm from being useful in practice, where time is an important factor. If time is not a factor, the reorganization algorithm returns an optimal assignment set. However, because time is an important factor in most real world cases, this reorganization algorithm is highly impractical except for the extremely small cases. Furthermore, this thesis provides an implementation for the two versions of reorganization algorithm (*CBF* and *DBF*) in a high-level simulator (*CROS*). In addition, this thesis provides a compilation of the results from testing the the implementation of the *CBF*.

As a result, some interesting design characteristics surfaced. This thesis covers a

few basic design characteristics that allows designers to design models such that the running time of the algorithm is close to the best case input, which is still exponential but smaller by an order of magnitude. Interestingly, this also opens up an entirely new area of research into the study of performance measures for models of OMACS. This thesis mentions briefly two such metrics from preliminary observation: efficiency in terms of reorganization algorithm performance, and flexibility in terms of how well a model is able to adapt to a changing environment [RDK06]. Another aspect for performance metrics is in the area of policies. Policies have varying effects on OMACS. This thesis briefly highlights two policies that have a profound effect on the time complexity of the reorganization algorithm.

Last but not least, subtle improvements can be made to reorganization algorithms if the algorithm is designed to be distributed among the agents. This thesis covers a brief overview of the complexity of communication among agents for distributed algorithms, as well as some of the gains in time complexity from adopting a distributed approach.

7.2 Future Work

This section lists several interesting areas for future work and improvements.

7.2.1 Extending the Model

Currently, OMACS provides limited functionality to reorganization algorithms. Particularly, the lack of information about how the internal workings of the `oaf()` and `rcf()` restrict the use of heuristics in search algorithms. Knowing that the `oaf()` and `rcf()` are deterministic is not sufficient. Without more information about the `oaf()` and `rcf()`, general purpose search algorithms cannot be implemented efficiently. As such, for any search algorithm to be efficient, the search algorithm will have to be implemented for a specific model with a specific `oaf()` and a specific `rcf()`. For example, there are two agents ($Agent1$ and $Agent2$), two roles ($Role1$ and $Role2$), one goal ($Goal1$), and one capability ($Capability1$). Both $Role1$ and $Role2$ requires

*Capability*1, and they achieve *Goal*1 with a score of 1.0. Both *Agent*1 and *Agent*2 possesses *Capability*1 with a score of 1.0. An `oaf()` might give an organization score higher than 2.0 if *Agent*1 is assigned to play both roles. Or an `oaf()` might give an organization score of 0.0 if *Agent*1 is assigned to play both roles.

Exposing the internal workings of the `oaf()` and `rcf()` require extensions to OMACS. With a good interface that is able to expose the internal workings of the `oaf()` and the `rcf()`, efficient algorithms can be designed for general use. For instance, for the `oaf()`, knowing that when two agents are playing the same role, their combined score would be a factor instead of the simple additive score would help tremendously in reducing the search space. Again, for the `rcf()`, knowing how much a capability contributes to the `rcf()` score would help. One such approach to exposing the internal workings of the `oaf()` and `rcf()` would be to encode the internal workings of the `oaf()` and `rcf()` into some standardized data structure. When an algorithm executes, that algorithm would then be able to extract information from the data structure about the `oaf()` and `rcf()`.

Further research into this area may reveal how the `oaf()` and `rcf()` can be redesigned for more effective uses.

7.2.2 Exploring Design Characteristics

The time complexity from Chapter 4 can be further improved. A finer grain time complexity will provide more information about the characteristics of OMACS with respect to the reorganization algorithms. With further understanding on the relationships between OMACS and design characteristics, and the relationships among the design characteristics, a metrics system can be introduced that would allow models to be evaluated on areas such as efficiency and flexibility.

7.2.3 Exploring Effects Of Policies

In § 6.3.2, some of the potential effects that assignment policies have on reorganization algorithms are mentioned. However, very little else is known about the types of effects

that different assignment policies can have on the time complexity of reorganization algorithms. Furthermore, knowing the effects of policies on OMACS (particularly the effects that policies have on reorganization algorithms) could also lead to design metrics for policies evaluation.

7.2.4 Distributing the Algorithm

§ 4.3 covered a brief overview of the communication costs for distributed reorganization algorithms. This thesis only breaches the surface of distributed algorithms. Even with the simplistic distributed version provided by this thesis, the time complexity improved in the best case. By adopting a purely distributed approach to designing reorganization algorithms, there could be significant gains in the time complexity as the number of agents increases. However, the additional communication costs could be significant as well. Further investigative research into this area may unveil the advantages and disadvantages of distributed algorithms for OMACS.

Bibliography

[BM01] K. S. Barber and C. E. Martin. Dynamic reorganization of decision-making groups. In *Proceedings of the fifth international conference on Autonomous agents*, pages 513–520, New York, NY, USA, 2001. ACM Press.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[CG99] Kathleen M. Carley and Les Gasser. Computational organization theory. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 299–330, Cambridge, MA, USA, 1999. MIT Press.

[Dav01] Paul Davidsson. Categories of artificial societies. In Andrea Omicini, Paolo Petta, and Robert Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *LNAI*, pages 1–9. Springer-Verlag, 2001.

[DD01] Virginia Dignum and Frank Dignum. Modelling agent societies: Co-ordination frameworks and institutions. In Pavel Brazdil and Alipio Jorge, editors, *Progress in Artificial Intelligence*, volume 2258 of *LNAI*, pages 191–204. Springer-Verlag, 2001.

[DDS04] Virginia Dignum, Frank Dignum, and Liz Sonenberg. Towards dynamic reorganization of agent societies. In *WORKSHOP on Coordination in Emergent Agent Societies*, 2004.

[DeL99] Scott A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems. In *Proc. of Workshop on Agent-Oriented Information Systems*, 1999.

[DeL02] Scott A. DeLoach. Modeling organizational rules in the multi-agent systems engineering methodology. In R. Cohen and B. Spencer, editors, *Advances in Artificial Intelligence: 15th Conference of the Canadian Society for Computational Studies of Intelligence*, volume 2338 of *LNAI*, pages 1–15. Springer-Verlag, 2002.

[DeL05] Scott A. DeLoach. Multiagent systems engineering of organization-based multiagent systems. In *SELMAS '05: Proceedings of the fourth international workshop on Software engineering for large-scale multi-agent systems*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[Dig04] Virginia Dignum. *A Model for Organizational Interaction: Based on Agents, Founded in Logic*. PhD thesis, Utrecht University, 2004.

[DM04] Scott A. DeLoach and Eric T. Matson. An organizational model for designing adaptive multiagent systems. In *Technical Report WS-04-02*, pages 66–73, San Jose, California, July 2004. The AAAI-04 Workshop on Agent Organizations: Theory and Practice (AOTP'04), AAAI Press.

[DM05] Scott A. DeLoach and Eric T. Matson. A capability based theory of artificial organizations. Working Draft, February 2005.

[DVSD04] Virginia Dignum, Javier Vázquez-Salceda, and Frank Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. In *PROMAS*, pages 181–198, 2004.

[EPS01] Marc Esteva, Julian Padget, and Carles Sierra. Formalizing a language for institutions and norms. In John-Jules C. Meyer and Milind Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *LNAI*, pages 348–366. Springer-Verlag, 2001.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society.

[FGJ+02] Jacques Ferber, Olivier Gutknecht, Catholijn M. Jonker, Jean-Pierre Müller, and Jan Treur. Organization models and behavioural requirements specification for multi-agent systems. In *Proc. of the 10th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'01*, LNAI. Springer-Verlag, 2002.

[FGM03] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *LNCS*, pages 214–230. Springer Berlin / Heidelberg, 2003.

[HL05] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.

[IGY92] Toru Ishida, Les Gasser, and Makoto Yokoo. Organization self-design of distributed production systems. *IEEE Transactions on Knowledge and Data Engineering*, 04(2):123–134, 1992.

[KG98] Stefan Kirn and Les Gasser. Organizational approaches to coordination in multi-agent systems. In *Special Issue on Intelligent Agents*, pages 23–29, 1998.

[ONL05] James Odell, Marian Nodine, and Renato Levy. A metamodel for agents, roles, and groups. In James Odell, Paolo Giorgini, and Jörg Müller, editors, *Agent-Oriented Software Engineering V: 5th International Workshop*, volume 3382 of *LNCS*, page 78. Springer Berlin / Heidelberg, 2005.

[RDK06] Robby, Scott A. DeLoach, and Valeriy A. Kolesnikov. Towards using design metrics for predicting system flexibility. In *Fundamental Approaches to Software Engineering (FASE'06)*, 2006.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[vLLD98] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, 1998.

[VSD03] Javier Vázquez-Salceda and Frank Dignum. Modelling electronic organizations. In Vladimir Marik, Jörg Müller, and Michal Pechoucek, editors, *Multi-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 584–593. Springer-Verlag, 2003.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[Zam02] Franco Zambonelli. Abstractions and infrastructures for the design and development of mobile agent organizations. In Michael J. Wooldridge, Gerhard Weiss, and Paolo Ciancarini, editors, *Agent-Oriented Software Engineering II*, volume 2222 of *LNCS*, pages 245–262. Springer-Verlag, 2002.

[ZCLL04] Haizheng Zhang, W. Bruce Croft, Brian Levine, and Victor Lesser. A multi-agent approach for peer-to-peer-based information retrieval systems. In *AAMAS2004*, 2004.